MSc in Data Science

NATIONAL CENTRE FOR SCIENTIFIC RESEARCH "DEMOKRITOS"

UNIVERSITY OF THE PELOPONNESE

Academic Year 2022 -23

Course: Deep Learning

Professor: Giannakopoulos Theodoros

Project Report

Subject: Comparative analysis between custom CNN trained from scratch, and pretrained face recognition models, using the ResNet50 CNN architecture.

*Additional demo scripts provided for live camera face recognition

Full Name: Charalampos Apostolakis **student ID.:** 2022202204004

Full Name: John Theocharis **student ID.:** 2022202204019

Date of Submission: 28/06/2023

➢ Introduction

General Objective

The objective is to develop accurate and robust face recognition models, comparing their capability of identifying individuals and demonstrate them using live camera face recognition. The final features of the fc layer are properly transformed to identify multiple classes (classification problem)(3 classes in our implementation). Every class represents a specific face or category, namely the first two classes are our faces, named “JohnT” and “HarisA” respectively and the 3rd class represents unknown faces and is named as “Unknown”. To this end 3 models are going to be developed and assessed:

- The 1st model is pretrained using the ResNet50 architecture and weights derived from IMAGENET1K_V2. The ImageNet dataset is a large-scale dataset widely used for training and evaluating deep learning models, particularly for image classification tasks. It contains millions of labeled images spanning over 1,000 different classes or categories. V2 refers to the specific version of the ImageNet dataset. The pre-trained weights capture the learned representations and knowledge from that training process. These weights are used as a starting point for transfer learning. In our analysis the first three out of five blocks are freezed (see image in the Introduction section) which in pytorch are described as “layers” but they include the 1st convolutional layer, the maxpool operation and the 2 following blocks. The last 2 blocks and the fully connected layer are trained from scratch using our own dataset.

- The 2nd model is deployed using the ResNet50 architecture but initialising the weights randomly (weights=None). The model will not have any prior knowledge or learned representations from pre-training on a large dataset like ImageNet. As a result, the model will start training from the beginning, using ResNet50 architecture and our own dataset.

- The 3rd model is a simple CNN entirely customised by ourselves. Its architecture will be described afterwards in the Implementation section. The model is trained only on our own dataset.

ResNet50 architecture

Within the framework of this project, among other CNN architectures we used the ResNet50 CNN model architecture.

ResNet-50 is a convolutional neural network (CNN) model that was introduced in the 2015 paper “Deep Residual Learning for Image Recognition” by He Kaiming, Zhang Xiangyu, Ren Shaoqing, and Sun Jian. CNNs are commonly used to power computer vision applications. It is part of the ResNet (Residual Network) family of models and has 50 layers. ResNet-50 is widely used for various computer vision tasks, including image classification, object detection, and image segmentation.

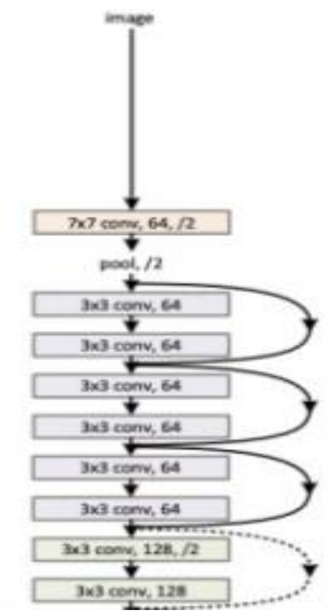
Residual neural networks are a type of artificial neural network (ANN) that forms networks by stacking residual blocks. The architecture of ResNet-50 introduces the concept of residual connections or skip connections, which address the problem of vanishing gradients during training of very deep neural networks. By using these skip connections, the model can learn residual functions, which are the differences between the input and the desired output.

Here is a high-level overview of the ResNet-50 architecture. It consists of 1 initial convolutional layer, followed by a MaxPool operation, 48 convolutional layers, and a fully connected layer after applying a global average pooling operation:

- Initial convolutional layer: This layer performs the initial convolution operation on the input image to extract low-level features.

- Four stages: ResNet-50 consists of four stages, each containing a set of convolutional blocks. The stages progressively downsample the spatial dimensions while increasing the number of filters.

- Convolutional blocks: Each stage consists of multiple convolutional blocks. Each block typically contains multiple convolutional layers followed by batch normalization and ReLU activation. The skip connections bypass these convolutional layers and add the input to the output, allowing the model to learn residual functions.



- Fully connected layer: At the end of the convolutional stages, a global average pooling operation is applied to reduce the spatial dimensions to a fixed size. This is followed by a fully connected layer that performs the final classification or regression based on the task at hand.

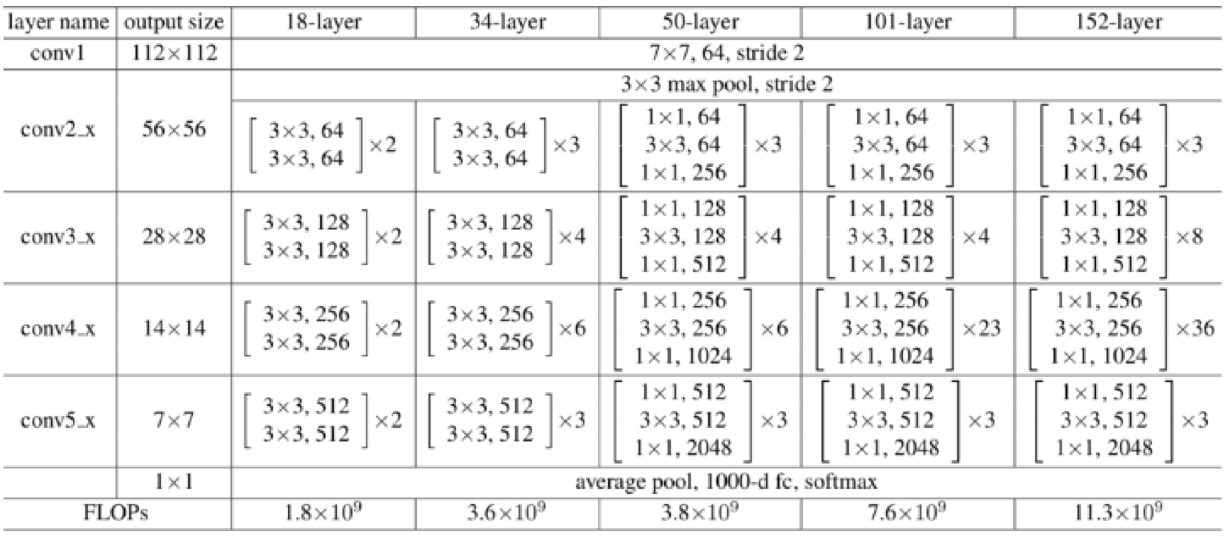
The 50-layer ResNet uses a bottleneck design for the building block. A bottleneck residual block uses 1×1 convolutions, known as a “bottleneck”, which reduces the number of parameters and matrix multiplications. This enables much faster training of each layer. It uses a stack of three layers rather than two layers.

The 50-layer ResNet architecture includes the following elements, as shown in the table below:

- A 7×7 kernel convolution alongside 64 other kernels with a 2-sized stride.
- A max pooling layer with a 2-sized stride.
- 9 more layers—3×3,64 kernel convolution, another with 1×1,64 kernels, and a third with 1×1,256 kernels. These 3 layers are repeated 3 times.
- 12 more layers with 1×1,128 kernels, 3×3,128 kernels, and 1×1,512 kernels, iterated 4 times.
- 18 more layers with 1×1,256 cores, and 2 cores 3×3,256 and 1×1,1024, iterated 6 times.

- 9 more layers with 1×1,512 cores, 3×3,512 cores, and 1×1,2048 cores iterated 3 times.
- (up to this point the network has 50 layers)
- Average pooling, followed by a fully connected layer with 1000 nodes, using the softmax activation function.

A tabular depiction of the aforementioned ResNet characteristics is presented below. We used the 50-layers model:



layer name	output size	18-layer	34-layer	50-layer	101-layer	152-layer
conv1	112×112	7×7, 64, stride 2				
conv2_x	56×56	3×3 max pool, stride 2				
		$\begin{bmatrix}3\times3, 64\\3\times3, 64\end{bmatrix}\times2$	$\begin{bmatrix}3\times3, 64\\3\times3, 64\end{bmatrix}\times3$	$\begin{bmatrix}1\times1, 64\\3\times3, 64\\1\times1, 256\end{bmatrix}\times3$	$\begin{bmatrix}1\times1, 64\\3\times3, 64\\1\times1, 256\end{bmatrix}\times3$	$\begin{bmatrix}1\times1, 64\\3\times3, 64\\1\times1, 256\end{bmatrix}\times3$
conv3_x	28×28	$\begin{bmatrix}3\times3, 128\\3\times3, 128\end{bmatrix}\times2$	$\begin{bmatrix}3\times3, 128\\3\times3, 128\end{bmatrix}\times4$	$\begin{bmatrix}1\times1, 128\\3\times3, 128\\1\times1, 512\end{bmatrix}\times4$	$\begin{bmatrix}1\times1, 128\\3\times3, 128\\1\times1, 512\end{bmatrix}\times4$	$\begin{bmatrix}1\times1, 128\\3\times3, 128\\1\times1, 512\end{bmatrix}\times8$
conv4_x	14×14	$\begin{bmatrix}3\times3, 256\\3\times3, 256\end{bmatrix}\times2$	$\begin{bmatrix}3\times3, 256\\3\times3, 256\end{bmatrix}\times6$	$\begin{bmatrix}1\times1, 256\\3\times3, 256\\1\times1, 1024\end{bmatrix}\times6$	$\begin{bmatrix}1\times1, 256\\3\times3, 256\\1\times1, 1024\end{bmatrix}\times23$	$\begin{bmatrix}1\times1, 256\\3\times3, 256\\1\times1, 1024\end{bmatrix}\times36$
conv5_x	7×7	$\begin{bmatrix}3\times3, 512\\3\times3, 512\end{bmatrix}\times2$	$\begin{bmatrix}3\times3, 512\\3\times3, 512\end{bmatrix}\times3$	$\begin{bmatrix}1\times1, 512\\3\times3, 512\\1\times1, 2048\end{bmatrix}\times3$	$\begin{bmatrix}1\times1, 512\\3\times3, 512\\1\times1, 2048\end{bmatrix}\times3$	$\begin{bmatrix}1\times1, 512\\3\times3, 512\\1\times1, 2048\end{bmatrix}\times3$
	1×1	average pool, 1000-d fc, softmax				
FLOPs		1.8×10^9	3.6×10^9	3.8×10^9	7.6×10^9	11.3×10^9

The ResNet-50 model has significantly improved the performance of deep neural networks on various computer vision benchmarks. It is pre-trained on large-scale datasets, such as ImageNet, and can be fine-tuned or used as a feature extractor for specific tasks.

Implementations of ResNet-50 are available in popular deep learning frameworks like TensorFlow and PyTorch, making it easy to use and adapt for different applications.

➢ Data Methodology

1. Data Collection:

In this step we collected a diverse and representative dataset of facial images. This dataset, comprised of face images, encompasses variations in lighting conditions, poses, facial expressions, ages, ethnicities, and gender. The dataset is obtained from 2 sources. The first part, obtained from Kaggle website is a dataset of human faces in order to train the model with various faces https://www.kaggle.com/datasets/ashwingupta3012/human-faces . The other part is comprised of our photos obtained from a special script using OpenCV library, initialising the live camera and taking random frames (this script will be described thoroughly in the implementation section). The overall dataset is comprised of approximately more than 7000 images per class (~21000 in total, considered to be a generally small, yet enough sized, for our low – scale project, dataset, limited proportionally to our computational resources).

2. Data Preprocessing and Augmentation:

Prior to training the ResNet-50 model, data preprocessing techniques are applied to enhance the quality and consistency of the dataset. This involves the following steps:

- ❖ Face Detection: Employing face detection algorithms to localize and extract facial regions from the images with specific dimensions.
- ❖ Image Resize and Normalization: Resize all images to the same size and perform normalization techniques
- ❖ Data Augmentation: Applying data augmentation methods, like random cropping, rotation, color scaling, and mirroring, to increase the dataset size and improve model generalization.

3. Dataset Split:

The dataset is divided into training, and validation. The training set is used for model training, the validation set is utilized for hyperparameter tuning and model selection. Due to GPU limitations, we didn’t perform concurrent hyperparameter tuning. We considered to tune our model with different batch size (the batch size determines how many samples are processed simultaneously before updating the model parameters based on the computed gradients), more training epochs, lower and higher learning rate, or maybe a different optimizer and activation function. We performed these tunings separately and statically with a much smaller dataset due to lack of computational resources. We ended up with determinate parameters that gave high accuracy (close to 100%), but we are not able to claim the same results in a much larger dataset. The specific parameters we decided to proceed with are analysed in the implementation phase. In terms of this project the validation set (20% of the total dataset), after deciding the parameters (with the aforementioned procedure) is used directly as a testing dataset.

4. Model Training:

The ResNet-50 architecture and our customised CNN architecture (2nd case) are employed for training the face recognition model. The preprocessed and augmented images are used as input to the model. The training process involves the following steps:

Transfer Learning (Only in the 1st case): Utilizing a pre-trained ResNet-50 model trained on a large-scale dataset (ImageNet) as the initial weights.

Fine-tuning (Only in the 1st case): Fine-tuning the pre-trained ResNet-50 model on the face recognition dataset to adapt it to the specific task.

Optimization: Selecting an appropriate optimization algorithm (e.g., stochastic gradient descent) and setting hyperparameters (learning rate, batch size, etc.) to train the model.

Loss Function: Employing a suitable loss function (in our case softmax loss following cross entropy criterion) to optimize the model's parameters for accurate face recognition.

5. Model Evaluation:

The trained model is evaluated on the testing dataset to assess its performance. The evaluation metrics include accuracy, loss, F1 score. The final results are presented in corresponding graphs to visualise the models’ performance evolution per epoch.

6. Live Demonstration, experimentation and assessment

Live Camera face recognition scripts are provided per model in order to demonstrate each model's capability to recognize correctly the face in front of the camera. As mentioned before there are 3 classes, 2 for our faces (JohnT, HarisA) and 1 (Unknown) for every other face. The code is developed in a dynamical form in order to be easily reproduced or expanded for example to more than 3 classes.

Apparently, live face recognition experiment/trial is the best way to access the models' performance, since the face recognition in continuous frames flow (video) is our ultimate goal (not just recognising a single photo, avoid overfitting and generalize better).

7. Conclusions

After careful consideration of all steps and results/observations and experiments, performance analysis is conducted to identify potential limitations, biases, or areas for improvement.

➢ Implementation and Guidelines

Passing from the data methodology analysis to the implementation phase in terms of creating the necessary models, we developed respective .py scripts, with the following order, as described below:

- face_capture
- random _face_preprocess
- augmentation_techniques_photos
- train_to_photos_ver4_pretrained
- train_to_photos_custom_multy
- live_recognition_ver_pretrained
- live_recognition_custom_cnn

*Important Note: We do not include separate script for no weights ResNet50 model (architecture only). In this case, the same script, as in the pretrained model, was used (train_to_photos_ver4_pretrained), just replacing the weights=ResNet50_Weights.IMAGENET1K_V2 with weights=None and saving the respective trained model state: "face_recognition_model_noweights_pretrained.pt".

Furthermore, in the project folder provided, we can find these additional files:

- face_recognition_custom_full_model.pt
- face_recognition_custom_model_dict.pt
- face_recognition_model_noweights_pretrained.pt
- face_recognition_model_V2weights_pretrained.pt
- haarcascade_frontalface_default

The first 4 files represent the saved trained models' state in order to be used by the live recognition .py scripts to perform live face recognition. As noticed for the customised model, we include two files, the full model's state and another file, only with a dictionary of parameters. The differences are presented below:

✓ Full Model State (full_model.pt):

a. Saving the model as a full model state allows you to save the entire state of the model, including all the parameters, optimizer state, and any additional attributes associated with the model.
b. This format is convenient when you want to resume training from the exact point where you left off, as it preserves the complete state of the model and optimizer.
c. When loading the model, you can simply load the model.pt file, and it will contain all the necessary information to continue training or perform inference.

✓ Dictionary of Model State (dict.pt):

a. Saving the model as a dictionary containing the model's state allows you to save only the parameters of the model without any additional information like optimizer state or other attributes.
b. This format is useful when you want to load the model's parameters into a different model architecture or when you only need to use the saved model for inference and don't need to continue training.
c. When loading the model, you need to manually create an instance of the desired model architecture and load the saved parameters using the dictionary.

In summary, saving the model as a full model state allows to save the complete state of the model and optimizer, which is useful for resuming training. Saving the model as a dictionary of model state **is more lightweight** and suitable when you only need to save the model's parameters for inference or when you want to transfer the parameters to a different model architecture.

As regards the "haarcascade_frontalface_default", it refers to a pre-trained Haar Cascade classifier for detecting frontal faces in images. It is a part of the OpenCV library, which provides a collection of pre-trained models for various computer vision tasks.

The Haar Cascade classifier is a machine learning-based approach for object detection in images. It works by analyzing patterns of intensity differences in the image to identify specific objects or features. The "frontalface_default" cascade specifically focuses on detecting frontal faces in various conditions, such as different poses, lighting conditions, and scales. It can be a quick and effective solution for face detection tasks, particularly in real-time applications or situations where real-time performance is desired. Therefore, it is considered suitable for our needs, by taking several frames of our faces using live camera, in order to gather a sufficient dataset of images for our own faces and train the models.

Moreover 3 .png images are included, depicting the Loss, f1 score and accuracy results per epoch for each one of the three models:

- training_metrics_custom_model
- training_metrics_noweights_pretrained
- training_metrics_V2weights_pretrained

Finally, the project folder includes 3 more folders:

- Photos: This folder contains the whole dataset of images (per class, represented by a separate folder) used to train the models.
- Test_folder: This folder contains a subset of the original dataset, with 100 images per class and it was used (due to computational limitations) for tuning purposes.
- live_face_recognition_with vectors: This is a compressed folder containing an initial trial to approach the live face recognition problem, using the “InceptionResnetV1” deep neural network architecture.
  Overall, InceptionResNetV1 combines the advantages of both Inception and ResNet architectures, leading to a model that can capture complex features at different scales while being relatively efficient to train. It has been widely used for various computer vision tasks, including image classification, object detection, and image segmentation.
  As noticed it seems to be more complex than the original ResNet models due to its multi-branch architecture and the incorporation of residual connections and more prone to overfitting, especially when working with limited training data.

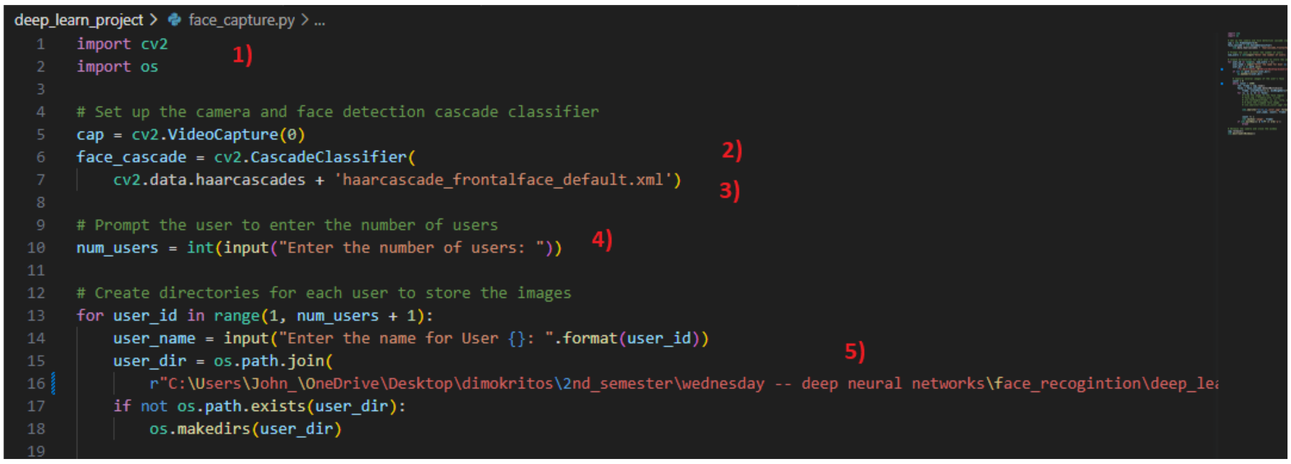
We fully deployed this model and although we didn’t use it to our analysis, it is considered a reliable choice and we valorised the knowledge obtained during its development process. Therefore, we included it to the project’s documentation.

A more detailed analysis as well as steps description for the main components of the projects’ implementation phase, mentioned before, is provided below:

1. face_capture

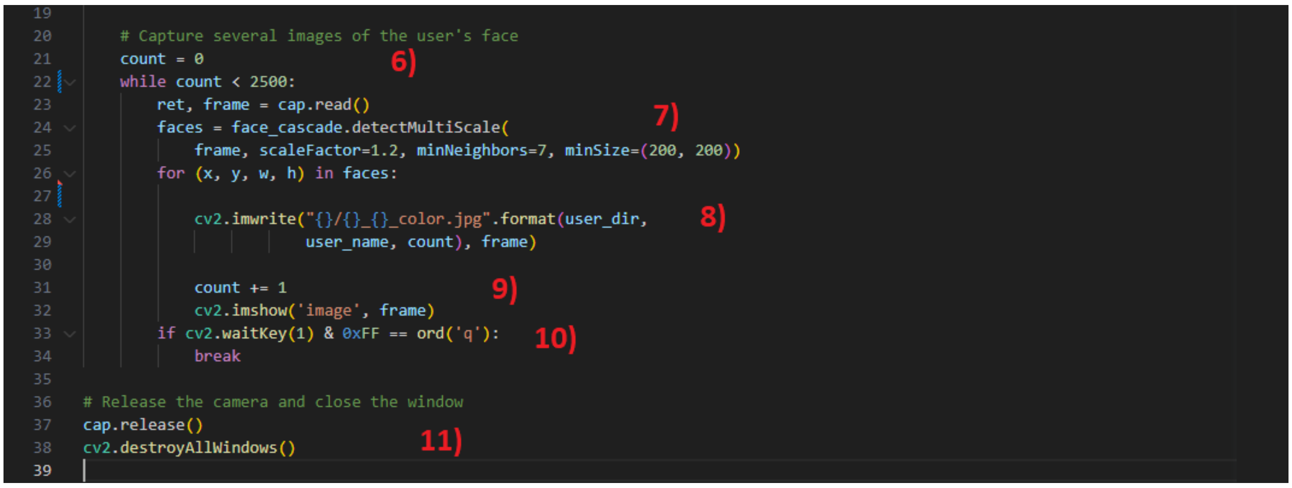
Overall this .py script utilizes the os and OpenCV libraries allowing the user to capture a specified number of images for multiple users' faces using a webcam and save them to individual directories. The implementation steps are described below:

1. Imports the necessary libraries: cv2 (OpenCV) and os.
2. Sets up the camera using cv2.VideoCapture(0), which initializes the webcam.
3. Loads the face detection cascade classifier using cv2.CascadeClassifier and the pre-trained XML file haarcascade_frontalface_default.xml. This classifier is used to detect faces in the captured frames (as mentioned before)
4. Prompts the user to enter the number of users (num_users) they want to capture images for.
5. Creates a directory for each user to store their images, naming the directory based on user’s input (prompt) The directory path is constructed using the user's name and a predefined base directory (r"C:\Users\User\Desktop\deep_learn_project\deep_learn_project\photos\aug"). This path can be modified according to individual preferences.



```python
import cv2
import os

# Set up the camera and face detection cascade classifier
cap = cv2.VideoCapture(0)
face_cascade = cv2.CascadeClassifier(
    cv2.data.haarcascades + 'haarcascade_frontalface_default.xml')

# Prompt the user to enter the number of users
num_users = int(input("Enter the number of users: "))

# Create directories for each user to store the images
for user_id in range(1, num_users + 1):
    user_name = input("Enter the name for User {}: ".format(user_id))
    user_dir = os.path.join(
        r"C:\Users\John_\OneDrive\Desktop\dimokritos\2nd_semester\wednesday -- deep neural networks\face_recogintion\deep_lea
    if not os.path.exists(user_dir):
        os.makedirs(user_dir)

```

6. Enters a loop that captures multiple images for each user. The loop runs until the required images are captured.
7. Inside the loop, it reads frames and calls the face_cascade.detectMultiScale function to detect faces in the frame. It returns a list of rectangles that represent the coordinates of detected faces.
8. For each detected face, the program saves the entire frame as an image using cv2.imwrite. The image file name includes the user's name, a count value, and "color" to indicate it is a color image.
9. The program displays the captured frame temporarily in a window using cv2.imshow.
10. The loop continues until the desired number of images is captured or until the user presses 'q' to quit.
11. After capturing images for all users, it releases the camera resource using cap.release() and closes the OpenCV window using cv2.destroyAllWindows().



```
    # Capture several images of the user's face
    count = 0
    while count < 2500:
        ret, frame = cap.read()
        faces = face_cascade.detectMultiScale(
            frame, scaleFactor=1.2, minNeighbors=7, minSize=(200, 200))
        for (x, y, w, h) in faces:

            cv2.imwrite("{}/{}_{}_color.jpg".format(user_dir,
                        user_name, count), frame)

            count += 1
            cv2.imshow('image', frame)
        if cv2.waitKey(1) & 0xFF == ord('q'):
            break

# Release the camera and close the window
cap.release()
cv2.destroyAllWindows()
```

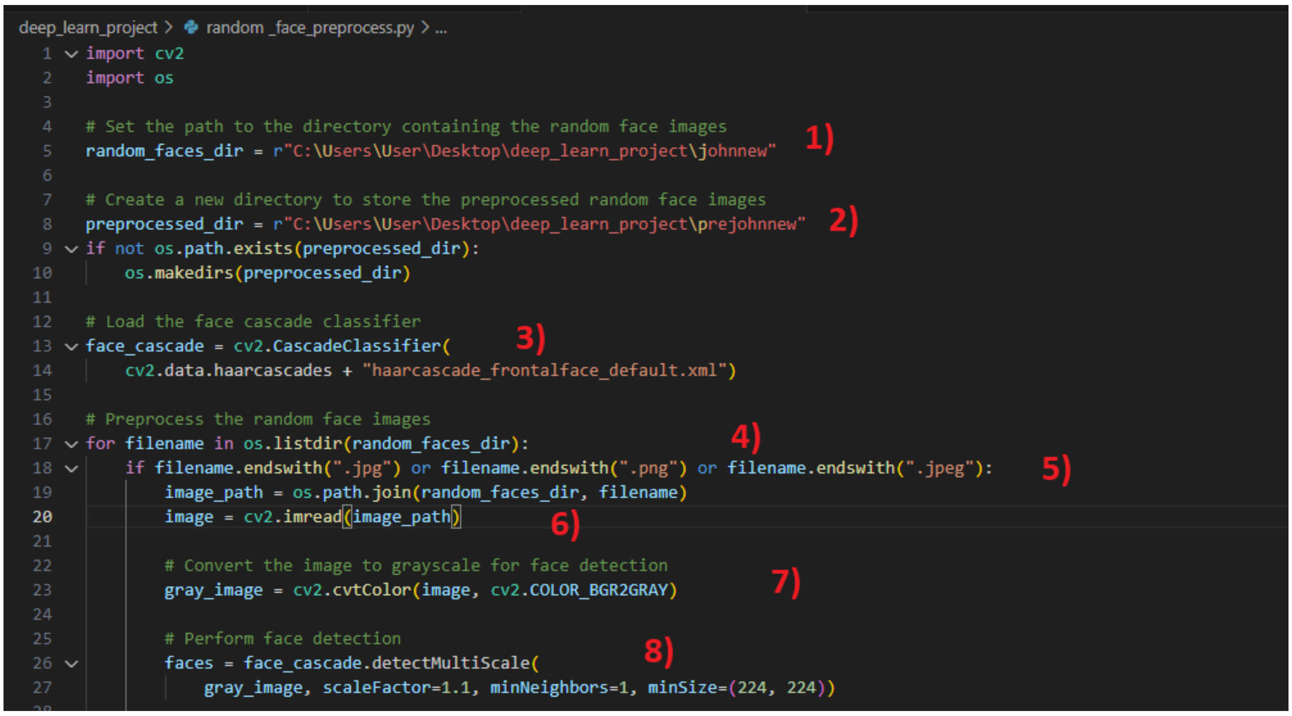
2. <u>random _face_preprocess</u>

Overall, this .py script reads certain types of face images (jpg, png, jpeg) from the directories created with the “face_capture” script or gathered by a different source, performs face detection and preprocessing, cropping and resizing the detected faces, and normalizing their pixel values, and finally saves the preprocessed face images to a new directory. The implementation steps are described below:

1. Sets the path to the directory containing the random face images. This could be the initial path set from the 1st step, or another path as per user’s

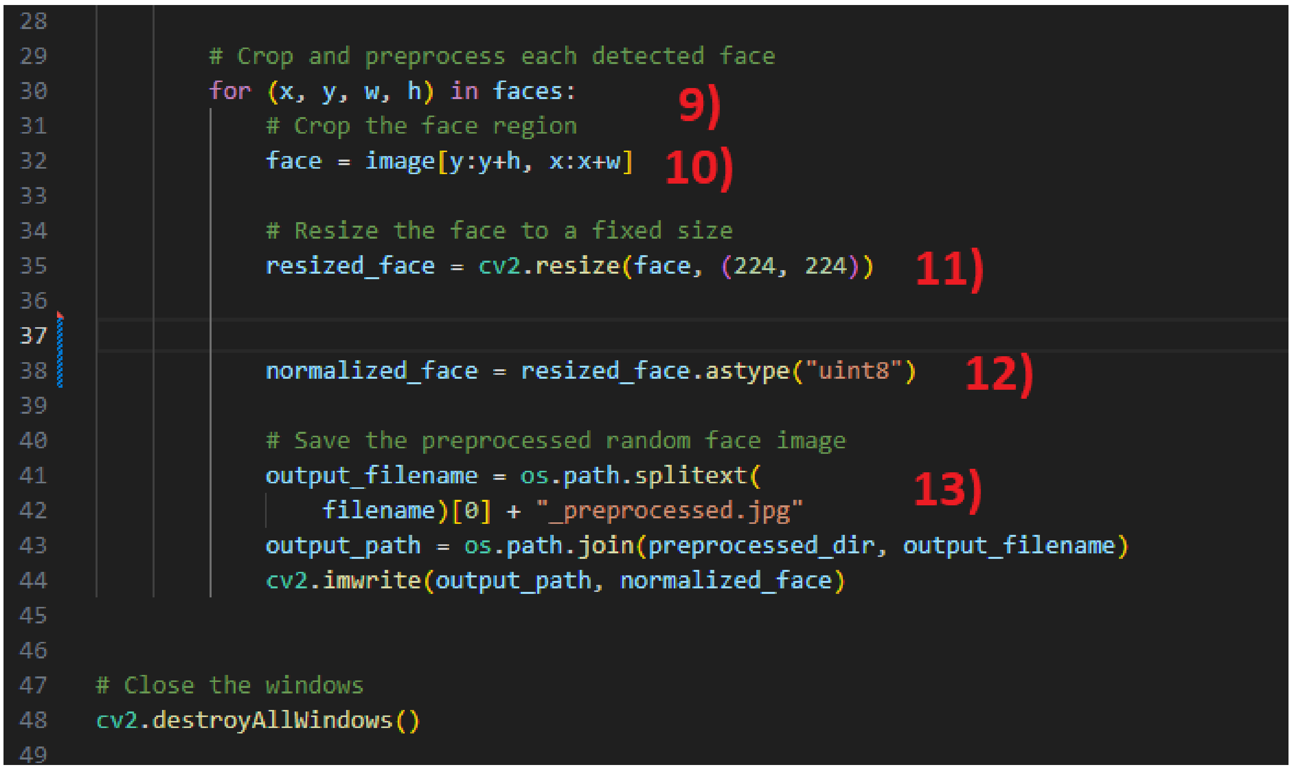
preference. We run this script iteratively, replacing the path for every directory of face images we are going to use.

2. Creates a new directory to store the preprocessed face images. The path is specified in the preprocessed_dir variable. If the directory doesn't exist, it is created using os.makedirs().
3. Loads the face cascade classifier using cv2.CascadeClassifier and the pre-trained XML file haarcascade_frontalface_default.xml. This classifier is used to detect faces in the images.
4. Iterates over each file in the random_faces_dir directory.
5. Checks if the file ends with ".jpg", ".png", or ".jpeg" using the endswith() method.
6. Reads the image using cv2.imread() by constructing the full path to the image.
7. Converts the image to grayscale using cv2.cvtColor() with the COLOR_BGR2GRAY conversion code.
8. Performs face detection on the grayscale image using face_cascade.detectMultiScale(). It returns a list of rectangles that represent the coordinates of detected faces.



```python
import cv2
import os

# Set the path to the directory containing the random face images
random_faces_dir = r"C:\Users\User\Desktop\deep_learn_project\johnnew"

# Create a new directory to store the preprocessed random face images
preprocessed_dir = r"C:\Users\User\Desktop\deep_learn_project\prejohnnew"
if not os.path.exists(preprocessed_dir):
    os.makedirs(preprocessed_dir)

# Load the face cascade classifier
face_cascade = cv2.CascadeClassifier(
    cv2.data.haarcascades + "haarcascade_frontalface_default.xml")

# Preprocess the random face images
for filename in os.listdir(random_faces_dir):
    if filename.endswith(".jpg") or filename.endswith(".png") or filename.endswith(".jpeg"):
        image_path = os.path.join(random_faces_dir, filename)
        image = cv2.imread(image_path)

        # Convert the image to grayscale for face detection
        gray_image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)

        # Perform face detection
        faces = face_cascade.detectMultiScale(
            gray_image, scaleFactor=1.1, minNeighbors=1, minSize=(224, 224))
```

9. Iterates over each detected face in the faces list.
10. Crops the face region from the original image using the coordinates of the detected face.
11. Resizes the cropped face to a fixed size of 224x224 pixels using cv2.resize().
12. Converts the normalized face image to the uint8 data type using astype("uint8").
13. Saves the preprocessed face image by constructing an output filename and path based on the original filename, appending "_preprocessed.jpg" to it, and using cv2.imwrite() to save the image.

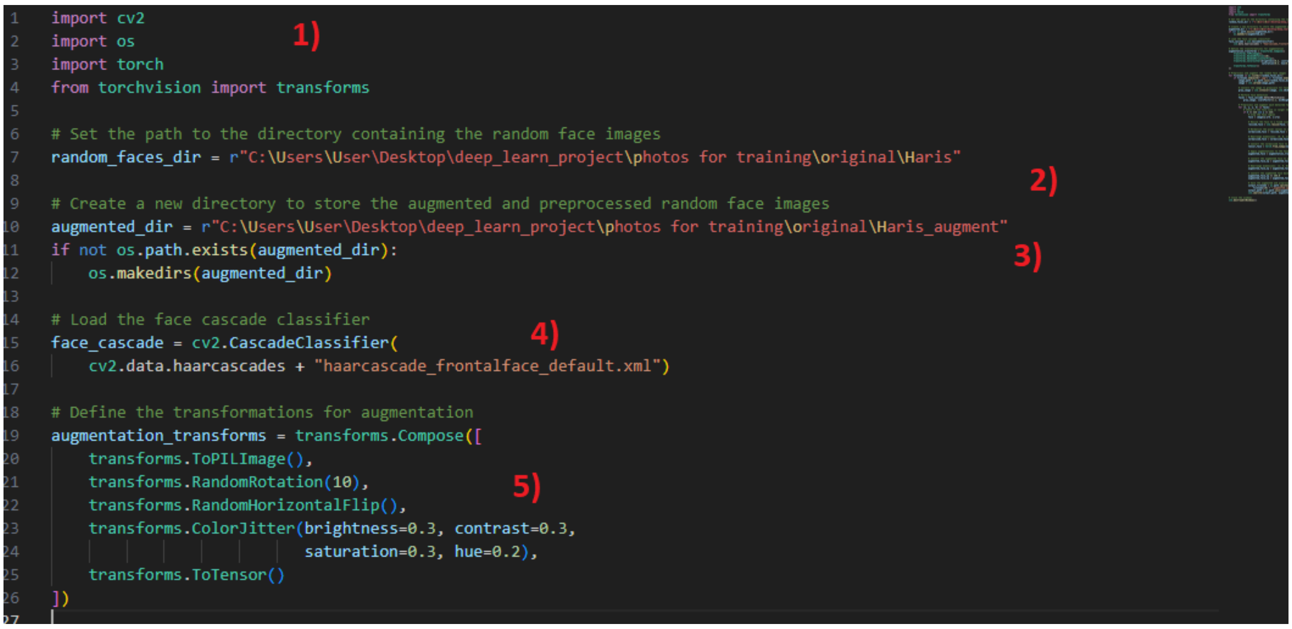


```python

        # Crop and preprocess each detected face
        for (x, y, w, h) in faces:
            # Crop the face region
            face = image[y:y+h, x:x+w]

            # Resize the face to a fixed size
            resized_face = cv2.resize(face, (224, 224))


            normalized_face = resized_face.astype("uint8")

            # Save the preprocessed random face image
            output_filename = os.path.splitext(
                filename)[0] + "_preprocessed.jpg"
            output_path = os.path.join(preprocessed_dir, output_filename)
            cv2.imwrite(output_path, normalized_face)


# Close the windows
cv2.destroyAllWindows()

```

3. <u>augmentation_techniques_photos</u>

Overall, this program applies face detection and data augmentation techniques to face images. It generates augmented versions of the original face images and saves them to a new directory. The final resulting augmented and preprocessed images dataset can be used for training our deep learning models. These transformations help the network generalize well to unseen data. By adding randomly generated image transformations in the training dataset, the network learns to be robust to variations in lighting conditions, contrast, and overall intensity, making it more effective at recognizing patterns in new, unseen images. Here's a breakdown of what the program does:

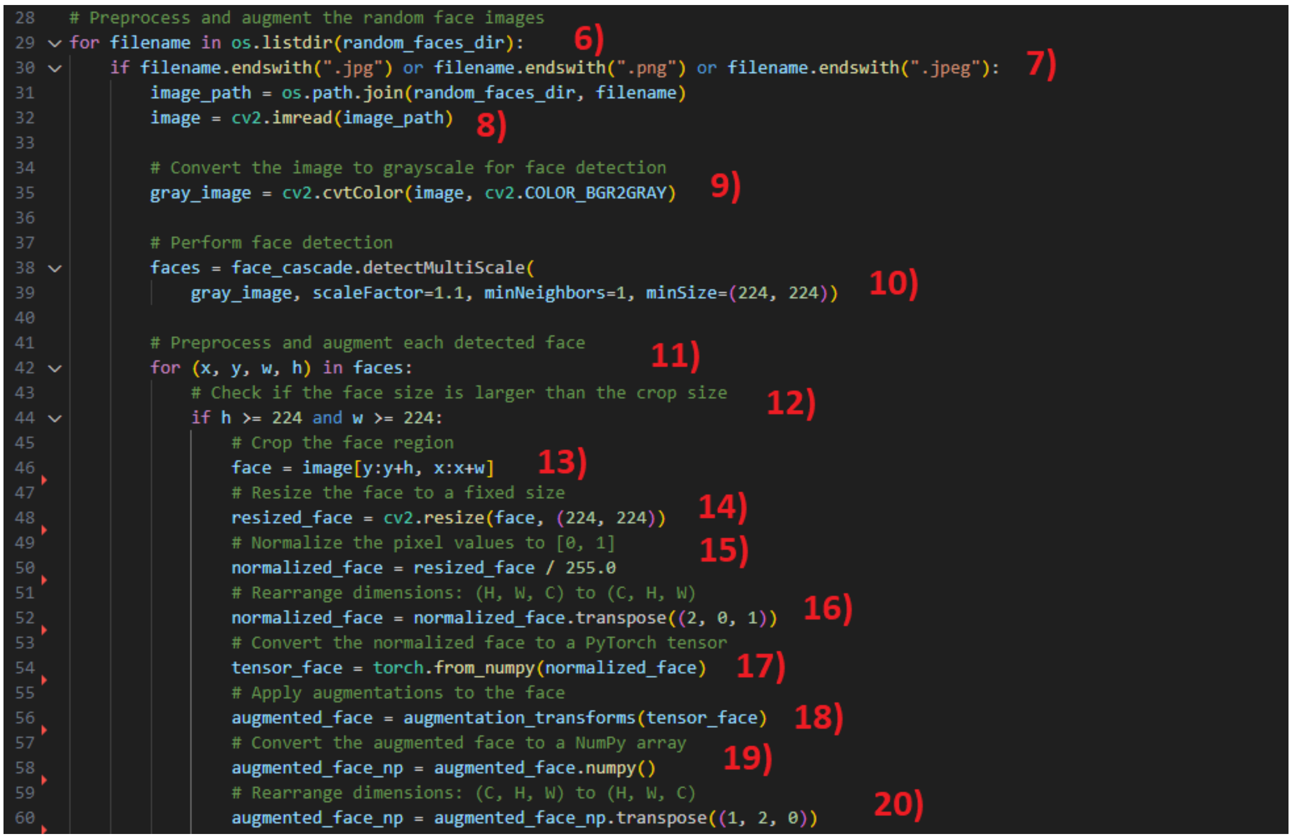
1. Imports the necessary libraries: cv2 (OpenCV), os, torch, and transforms from torchvision.
2. Sets the path to the desirable directory, where we aim to perform augmentation.
3. Creates a new directory to store the augmented and original face images. The path is specified in the augmented_dir variable. If the directory doesn't exist, it is created using os.makedirs().
4. Loads the face cascade classifier using cv2.CascadeClassifier and the pre-trained XML file haarcascade_frontalface_default.xml. This classifier is used to detect faces in the images.
5. Defines a series of transformations for data augmentation using the transforms.Compose() function from torchvision. The transformations include converting the image to a PIL image, random rotation, random horizontal flip, color jitter, and converting the image to a PyTorch tensor.



```python
import cv2
import os
import torch
from torchvision import transforms

# Set the path to the directory containing the random face images
random_faces_dir = r"C:\Users\User\Desktop\deep_learn_project\photos for training\original\Haris"

# Create a new directory to store the augmented and preprocessed random face images
augmented_dir = r"C:\Users\User\Desktop\deep_learn_project\photos for training\original\Haris_augment"
if not os.path.exists(augmented_dir):
    os.makedirs(augmented_dir)

# Load the face cascade classifier
face_cascade = cv2.CascadeClassifier(
    cv2.data.haarcascades + "haarcascade_frontalface_default.xml")

# Define the transformations for augmentation
augmentation_transforms = transforms.Compose([
    transforms.ToPILImage(),
    transforms.RandomRotation(10),
    transforms.RandomHorizontalFlip(),
    transforms.ColorJitter(brightness=0.3, contrast=0.3,
                           saturation=0.3, hue=0.2),
    transforms.ToTensor()
])
```

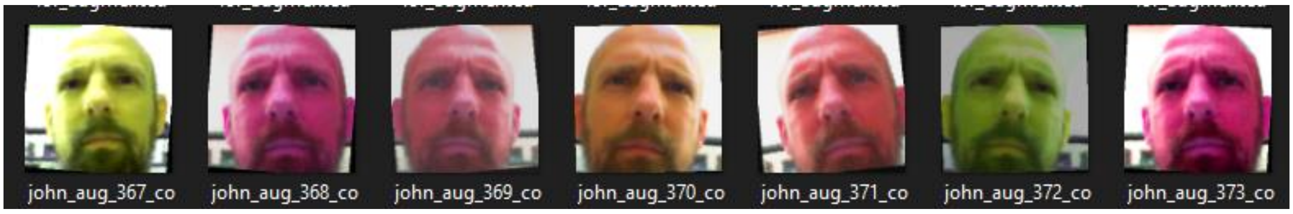
6. Iterates over each file in the random_faces_dir directory.
7. Checks if the file ends with ".jpg", ".png", or ".jpeg" using the endswith() method.
8. Reads the image using cv2.imread() by constructing the full path to the image.
9. Converts the image to grayscale using cv2.cvtColor() with the COLOR_BGR2GRAY conversion code.
10. Performs face detection on the grayscale image using face_cascade.detectMultiScale(). It returns a list of rectangles that represent the coordinates of detected faces.

11. Preprocesses and augments each detected face.
12. Checks if the detected face size is larger than or equal to 224x224 pixels.
13. Crops the face region from the original image using the coordinates of the detected face.
14. Resizes the cropped face to a fixed size of 224x224 pixels using cv2.resize().
15. Normalizes the pixel values of the resized face image to be in the range [0, 1] by dividing by 255.0.
16. Rearranges the dimensions of the normalized face image from (H, W, C) to (C, H, W).
17. Converts the normalized face image to a PyTorch tensor using torch.from_numpy().
18. Applies the defined augmentation transformations to the face tensor.
19. Converts the augmented face tensor to a NumPy array using .numpy().
20. Rearranges the dimensions of the augmented face array from (C, H, W) to (H, W, C).
21. Saves the augmented and preprocessed face image by constructing an output filename and path based on the original filename, appending "_augmented.jpg" to it, and using cv2.imwrite() to save the image.



```python
# Preprocess and augment the random face images
for filename in os.listdir(random_faces_dir):
    if filename.endswith(".jpg") or filename.endswith(".png") or filename.endswith(".jpeg"):
        image_path = os.path.join(random_faces_dir, filename)
        image = cv2.imread(image_path)

        # Convert the image to grayscale for face detection
        gray_image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)

        # Perform face detection
        faces = face_cascade.detectMultiScale(
            gray_image, scaleFactor=1.1, minNeighbors=1, minSize=(224, 224))

        # Preprocess and augment each detected face
        for (x, y, w, h) in faces:
            # Check if the face size is larger than the crop size
            if h >= 224 and w >= 224:
                # Crop the face region
                face = image[y:y+h, x:x+w]
                # Resize the face to a fixed size
                resized_face = cv2.resize(face, (224, 224))
                # Normalize the pixel values to [0, 1]
                normalized_face = resized_face / 255.0
                # Rearrange dimensions: (H, W, C) to (C, H, W)
                normalized_face = normalized_face.transpose((2, 0, 1))
                # Convert the normalized face to a PyTorch tensor
                tensor_face = torch.from_numpy(normalized_face)
                # Apply augmentations to the face
                augmented_face = augmentation_transforms(tensor_face)
                # Convert the augmented face to a NumPy array
                augmented_face_np = augmented_face.numpy()
                # Rearrange dimensions: (C, H, W) to (H, W, C)
                augmented_face_np = augmented_face_np.transpose((1, 2, 0))
```

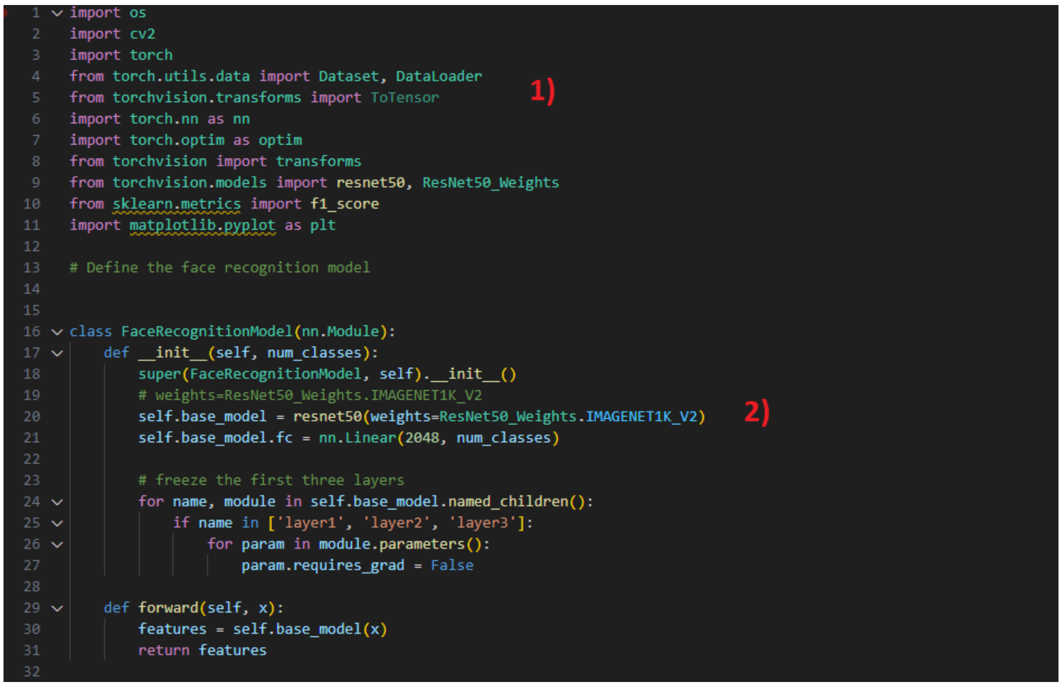
The final product of the augmented photos is shown in the figure bellow



4. <u>**train_to_photos_ver4_pretrained**</u>

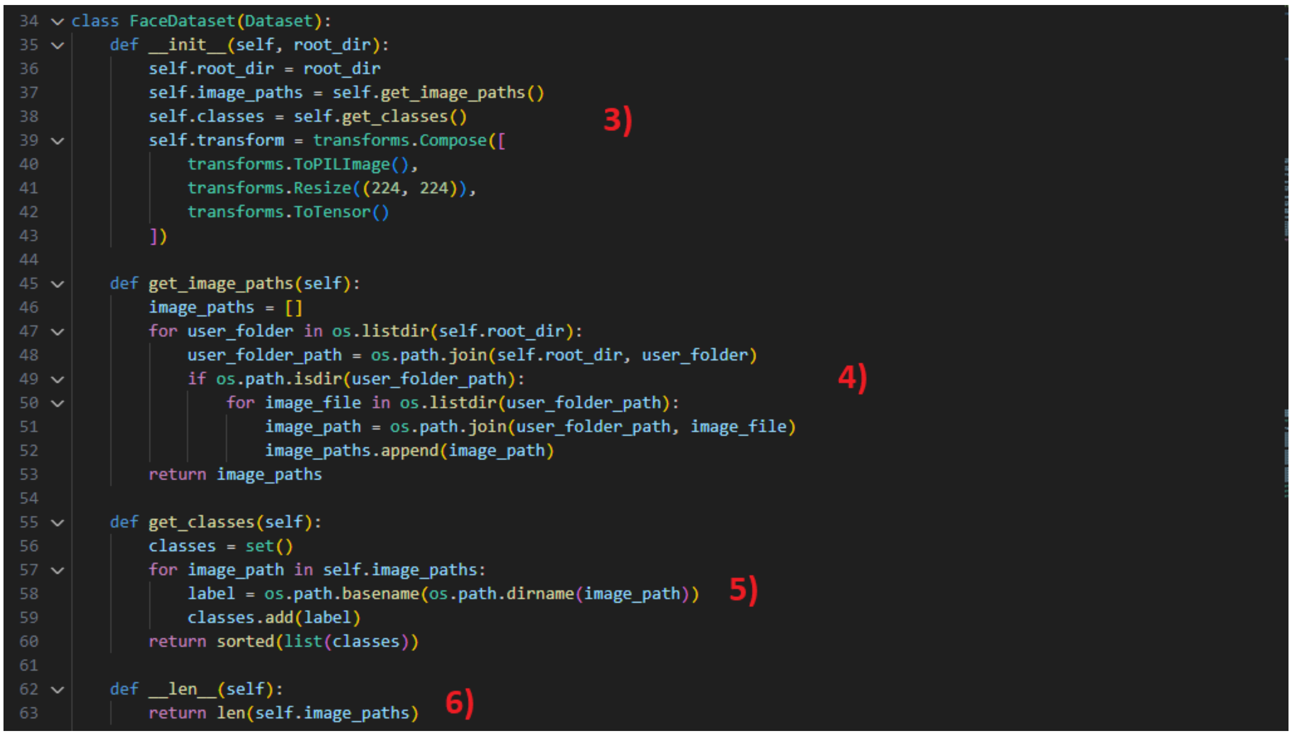
This .py script is used to train convolutional neural network (CNN) based on the ResNet-50 architecture and weights derived from IMAGENET1K_V2, as described in the data methodology section. These weights are used as a starting point for transfer learning. The first 3/5 blocks are used as such (freeze) but the last 2 blocks and the fully connected layer are trained from scratch using our own dataset. Here's a breakdown of what the program does:

1. It imports the necessary libraries, including os, cv2, torch, torchvision, sklearn and matplotlib.
2. It defines a CNN model called FaceRecognitionModel that is based on the ResNet-50 architecture. The model is initialized with a specified number of output classes for face recognition. It inherits from the nn.Module class, which is the base class for all neural network models in PyTorch. Inside the constructor (__init__ method), it initializes the ResNet-50 base model and **replaces its fully connected layer (fc) with a new linear layer that has num_classes output units**. Additionally, it freezes the parameters of the first three layers (layer1, layer2, and layer3) to prevent them from being updated during training. The forward method defines the forward pass of the model, where input x is passed through the base model and the features are returned.



```python
import os
import cv2
import torch
from torch.utils.data import Dataset, DataLoader
from torchvision.transforms import ToTensor
import torch.nn as nn
import torch.optim as optim
from torchvision import transforms
from torchvision.models import resnet50, ResNet50_Weights
from sklearn.metrics import f1_score
import matplotlib.pyplot as plt

# Define the face recognition model


class FaceRecognitionModel(nn.Module):
    def __init__(self, num_classes):
        super(FaceRecognitionModel, self).__init__()
        # weights=ResNet50_Weights.IMAGENET1K_V2
        self.base_model = resnet50(weights=ResNet50_Weights.IMAGENET1K_V2)
        self.base_model.fc = nn.Linear(2048, num_classes)

        # freeze the first three layers
        for name, module in self.base_model.named_children():
            if name in ['layer1', 'layer2', 'layer3']:
                for param in module.parameters():
                    param.requires_grad = False

    def forward(self, x):
        features = self.base_model(x)
        return features

```

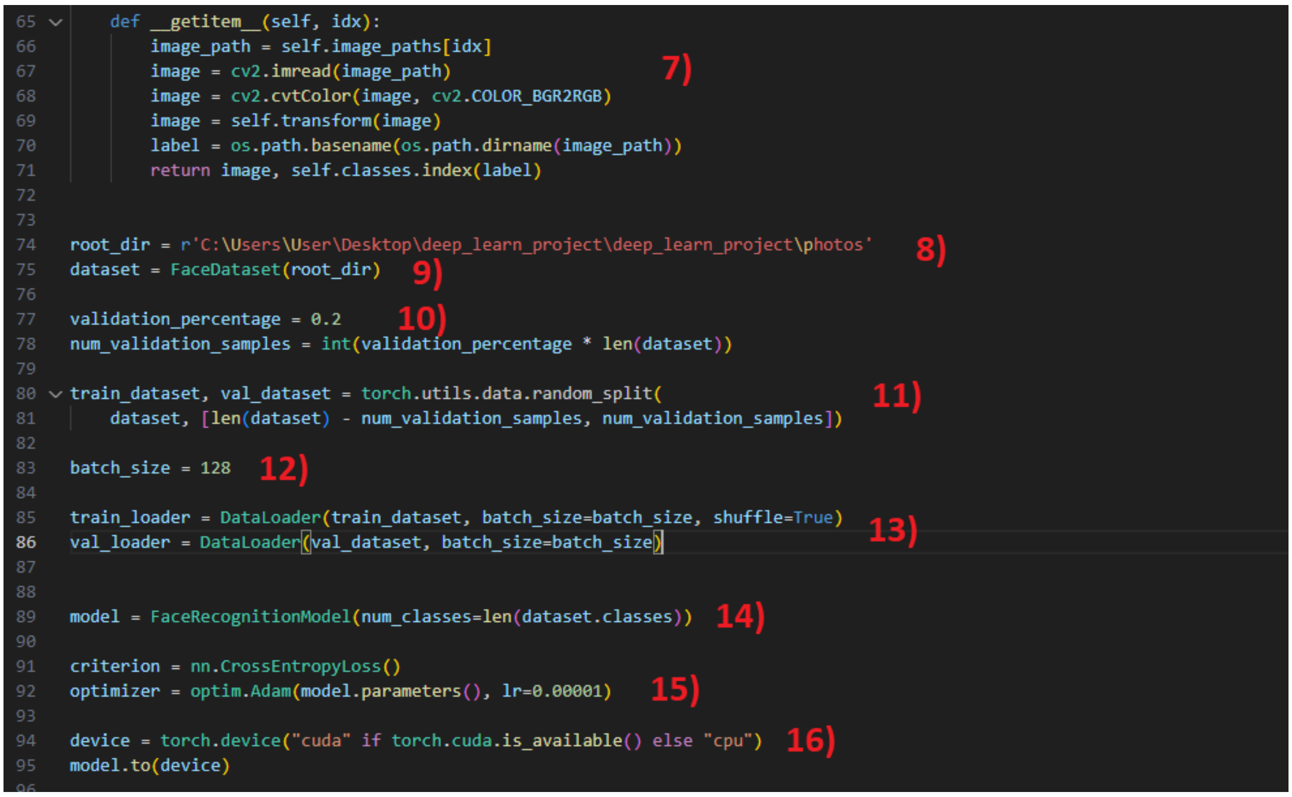
3. It defines a custom dataset called FaceDataset that represents the face images dataset. It loads the images from a specified root directory, preprocesses them by resizing and converting to tensors, and assigns labels based on the folder structure. It inherits from the Dataset class provided by PyTorch. Inside the constructor, it initializes the root directory, gets the image paths from the root directory, retrieves the unique classes from the image paths, and defines a series of image transformations using transforms.Compose.
4. The get_image_paths method iterates over the root directory, finds the image files, and appends their paths to a list.
5. The get_classes method extracts the class labels from the image paths and returns them as a sorted list.
6. The __len__ method returns the total number of images in the dataset.



```python
class FaceDataset(Dataset):
    def __init__(self, root_dir):
        self.root_dir = root_dir
        self.image_paths = self.get_image_paths()
        self.classes = self.get_classes()
        self.transform = transforms.Compose([
            transforms.ToPILImage(),
            transforms.Resize((224, 224)),
            transforms.ToTensor()
        ])

    def get_image_paths(self):
        image_paths = []
        for user_folder in os.listdir(self.root_dir):
            user_folder_path = os.path.join(self.root_dir, user_folder)
            if os.path.isdir(user_folder_path):
                for image_file in os.listdir(user_folder_path):
                    image_path = os.path.join(user_folder_path, image_file)
                    image_paths.append(image_path)
        return image_paths

    def get_classes(self):
        classes = set()
        for image_path in self.image_paths:
            label = os.path.basename(os.path.dirname(image_path))
            classes.add(label)
        return sorted(list(classes))

    def __len__(self):
        return len(self.image_paths)
```

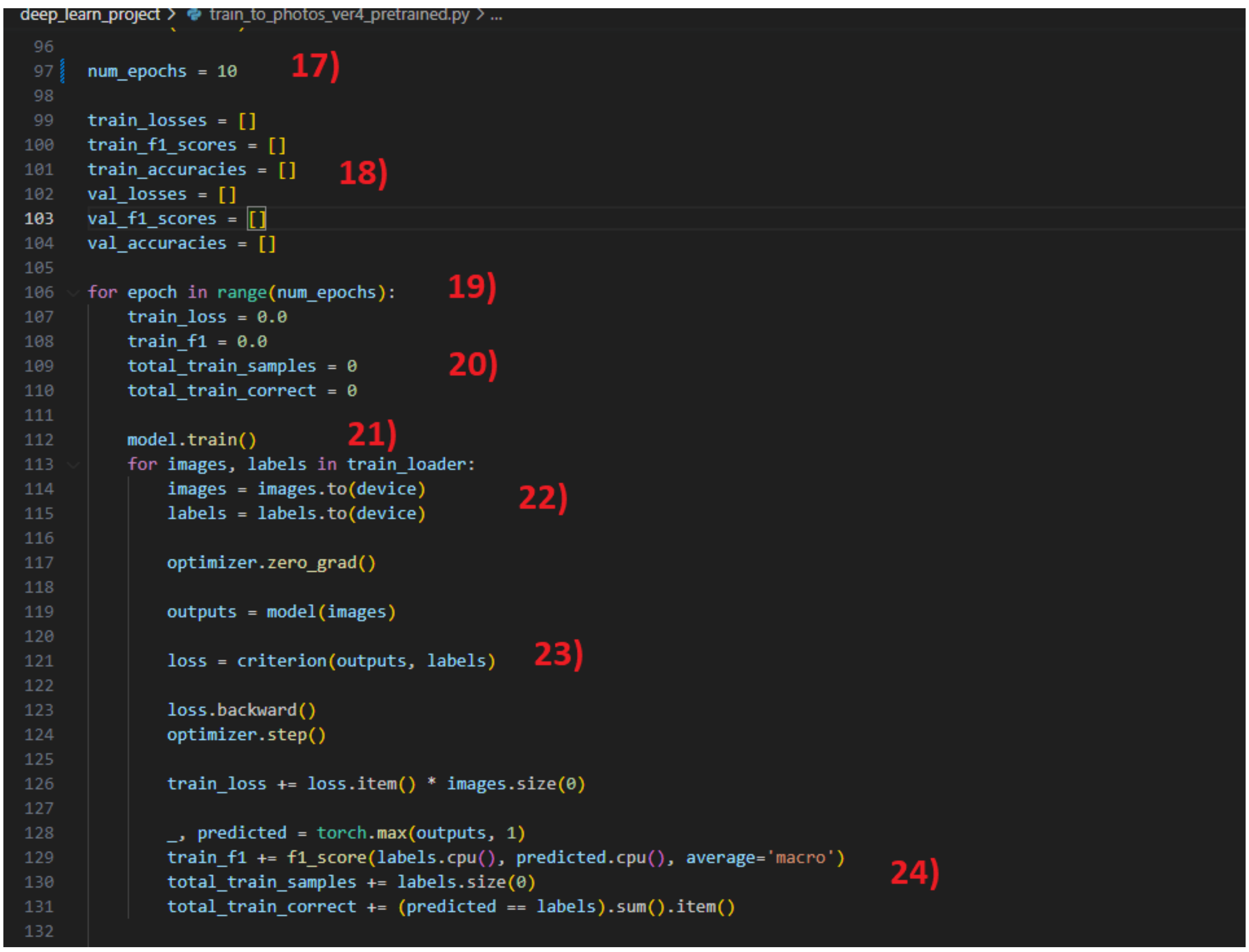
7. The __getitem__ method retrieves an image and its corresponding label at a given index. It reads the image using OpenCV, converts it to RGB color space, applies the defined transformations, and returns the transformed image along with the index of the corresponding label in the list of classes.
8. Sets the root directory path where the face images are located.
9. Creates an instance of the FaceDataset class by passing the root directory path. This dataset will be used for training and validation.
10. Defines the validation percentage (e.g., 0.2) and calculates the number of validation samples based on the dataset size.
11. Splits the dataset into training and validation datasets using torch.utils.data.random_split. The training dataset contains (1 - validation_percentage) portion of the data, and the validation dataset contains validation_percentage portion.
    *As described in data methodology, validation and tuning of hyperparameters was done asynchronously and separately with a smaller dataset due to computational limitations, so in this case validation set is used directly as testing set.
12. Sets the batch size for training and validation loaders to 128.
13. Creates DataLoader instances for the training and validation datasets. These loaders provide batches of data during training and evaluation.

14. Creates an instance of the FaceRecognitionModel class, specifying the number of classes based on the len(dataset.classes).
15. Defines the loss function (nn.CrossEntropyLoss) and the optimizer (optim.Adam) with learning rate = 0.00001 for training the model.
16. Checks if a GPU is available and moves the model to the GPU if possible.



```
    def __getitem__(self, idx):
        image_path = self.image_paths[idx]
        image = cv2.imread(image_path)
        image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
        image = self.transform(image)
        label = os.path.basename(os.path.dirname(image_path))
        return image, self.classes.index(label)


root_dir = r'C:\Users\User\Desktop\deep_learn_project\deep_learn_project\photos'
dataset = FaceDataset(root_dir)

validation_percentage = 0.2
num_validation_samples = int(validation_percentage * len(dataset))

train_dataset, val_dataset = torch.utils.data.random_split(
    dataset, [len(dataset) - num_validation_samples, num_validation_samples])

batch_size = 128

train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
val_loader = DataLoader(val_dataset, batch_size=batch_size)


model = FaceRecognitionModel(num_classes=len(dataset.classes))

criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.00001)

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model.to(device)
```

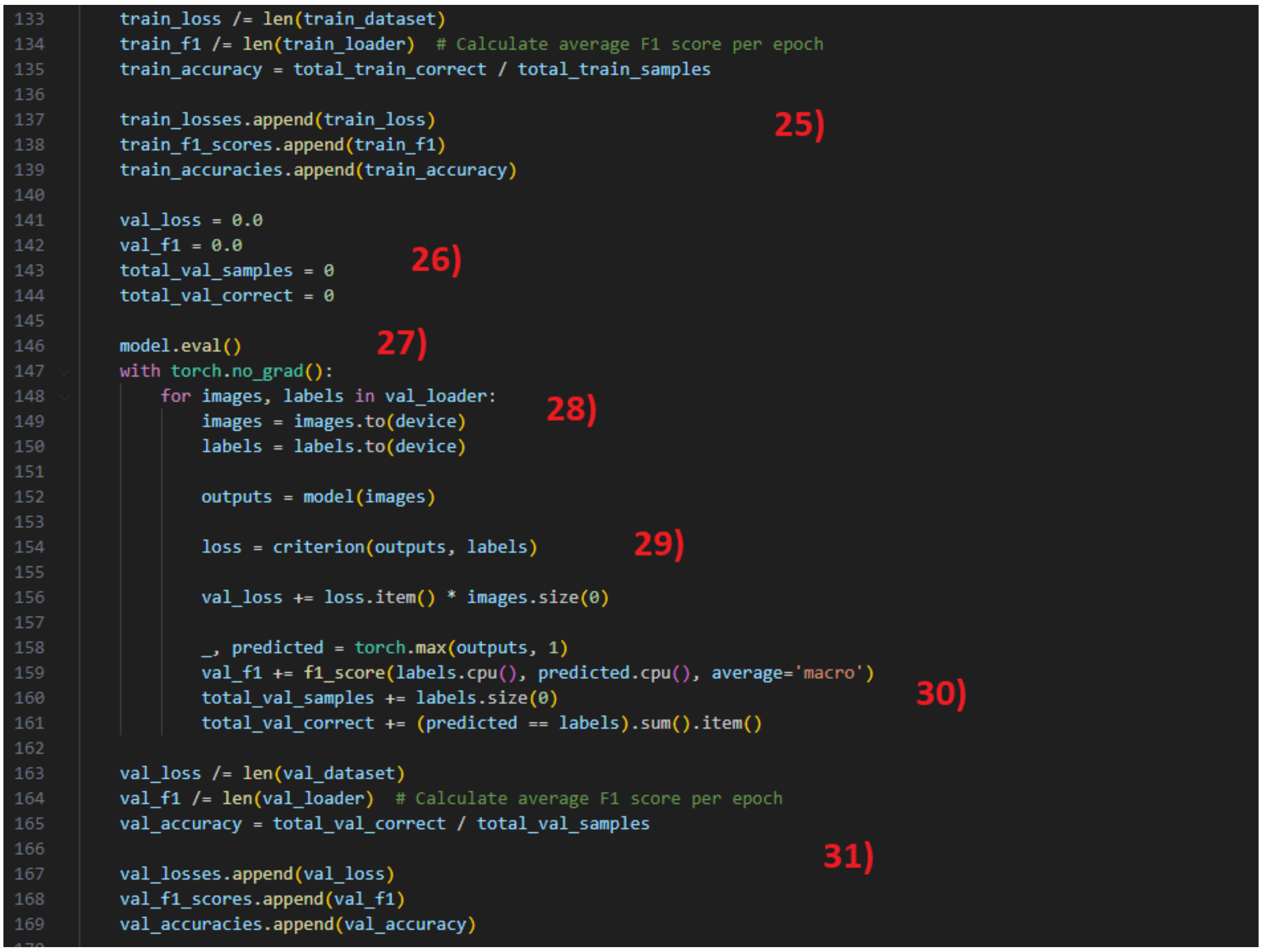
17. Specifies the number of epochs for training.
18. Initializes empty lists to store training and validation losses, F1 scores, and accuracies.
19. Starts the training loop over the specified number of epochs (5 in our case).
20. Initializes variables for tracking training loss, F1 score, number of correct predictions, and total samples.
21. Sets the model in training mode.
22. Iterates over batches of images and labels in the training loader.
23. Performs a forward pass through the model, calculates the loss, and updates the model's parameters.
24. Tracks the training loss, F1 score, number of correct predictions, and total samples for computing metrics.



deep_learn_project > train_to_photos_ver4_pretrained.py > ...

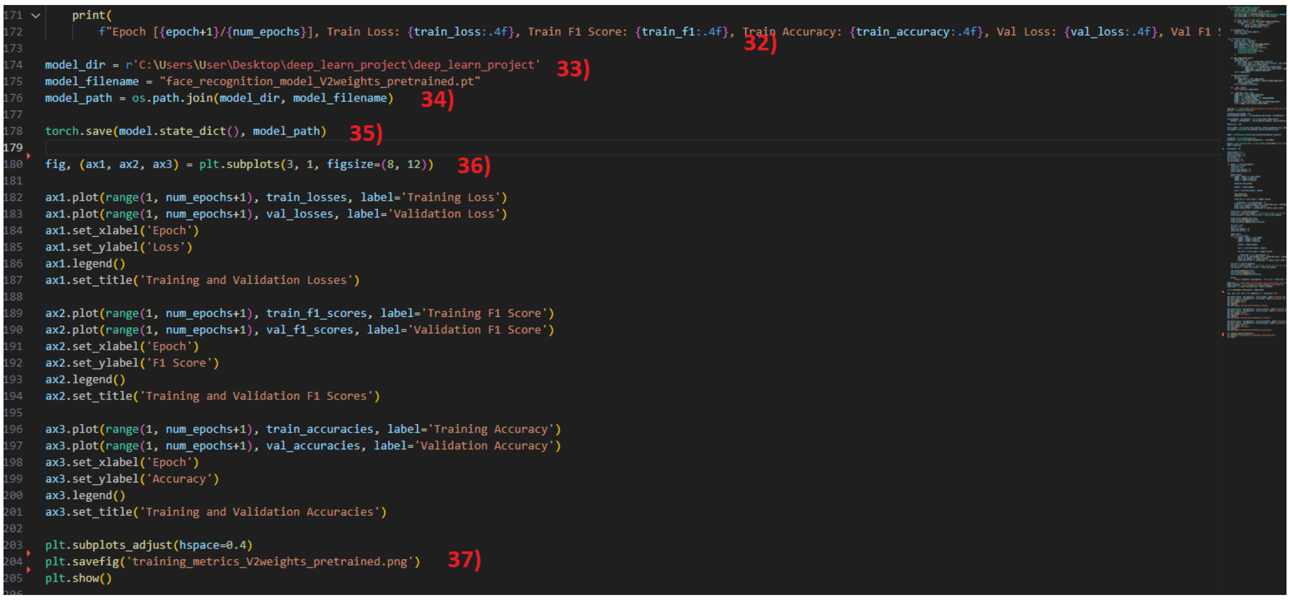
```
num_epochs = 10          17)

train_losses = []
train_f1_scores = []
train_accuracies = []     18)
val_losses = []
val_f1_scores = []
val_accuracies = []

for epoch in range(num_epochs):          19)
    train_loss = 0.0
    train_f1 = 0.0
    total_train_samples = 0              20)
    total_train_correct = 0

    model.train()          21)
    for images, labels in train_loader:
        images = images.to(device)            22)
        labels = labels.to(device)

        optimizer.zero_grad()

        outputs = model(images)

        loss = criterion(outputs, labels)      23)

        loss.backward()
        optimizer.step()

        train_loss += loss.item() * images.size(0)

        _, predicted = torch.max(outputs, 1)
        train_f1 += f1_score(labels.cpu(), predicted.cpu(), average='macro')
        total_train_samples += labels.size(0)                                   24)
        total_train_correct += (predicted == labels).sum().item()
```

25. Calculates and stores the average training loss, F1 score, and accuracy for the epoch.
26. Initializes variables for tracking validation loss, F1 score, number of correct predictions, and total samples.
27. Sets the model in evaluation mode (no gradient calculations).
28. Iterates over batches of images and labels in the validation loader.
29. Performs a forward pass through the model and calculates the loss for validation.
30. Tracks the validation loss, F1 score, number of correct predictions, and total samples for computing metrics.
31. Calculates and stores the average validation loss, F1 score, and accuracy for the epoch.



```python
    train_loss /= len(train_dataset)
    train_f1 /= len(train_loader)  # Calculate average F1 score per epoch
    train_accuracy = total_train_correct / total_train_samples

    train_losses.append(train_loss)
    train_f1_scores.append(train_f1)
    train_accuracies.append(train_accuracy)

    val_loss = 0.0
    val_f1 = 0.0
    total_val_samples = 0
    total_val_correct = 0

    model.eval()
    with torch.no_grad():
        for images, labels in val_loader:
            images = images.to(device)
            labels = labels.to(device)

            outputs = model(images)

            loss = criterion(outputs, labels)

            val_loss += loss.item() * images.size(0)

            _, predicted = torch.max(outputs, 1)
            val_f1 += f1_score(labels.cpu(), predicted.cpu(), average='macro')
            total_val_samples += labels.size(0)
            total_val_correct += (predicted == labels).sum().item()

    val_loss /= len(val_dataset)
    val_f1 /= len(val_loader)  # Calculate average F1 score per epoch
    val_accuracy = total_val_correct / total_val_samples

    val_losses.append(val_loss)
    val_f1_scores.append(val_f1)
    val_accuracies.append(val_accuracy)
```

32. Prints the metrics for the current epoch. The procedure is repeated until the end of the loop.
33. Sets the directory path and filename for saving the trained model.
34. Constructs the path to save the model.
35. It saves the trained model to a specified file path using torch.save.
36. It plots the training and testing metrics (loss, F1 score, and accuracy) over the epochs using matplotlib.
37. It saves the plot as an image file and displays it.



```
    print(
        f"Epoch [{epoch+1}/{num_epochs}], Train Loss: {train_loss:.4f}, Train F1 Score: {train_f1:.4f}, Train Accuracy: {train_accuracy:.4f}, Val Loss: {val_loss:.4f}, Val F1

model_dir = r'C:\Users\User\Desktop\deep_learn_project\deep_learn_project'
model_filename = "face_recognition_model_V2weights_pretrained.pt"
model_path = os.path.join(model_dir, model_filename)

torch.save(model.state_dict(), model_path)

fig, (ax1, ax2, ax3) = plt.subplots(3, 1, figsize=(8, 12))

ax1.plot(range(1, num_epochs+1), train_losses, label='Training Loss')
ax1.plot(range(1, num_epochs+1), val_losses, label='Validation Loss')
ax1.set_xlabel('Epoch')
ax1.set_ylabel('Loss')
ax1.legend()
ax1.set_title('Training and Validation Losses')

ax2.plot(range(1, num_epochs+1), train_f1_scores, label='Training F1 Score')
ax2.plot(range(1, num_epochs+1), val_f1_scores, label='Validation F1 Score')
ax2.set_xlabel('Epoch')
ax2.set_ylabel('F1 Score')
ax2.legend()
ax2.set_title('Training and Validation F1 Scores')

ax3.plot(range(1, num_epochs+1), train_accuracies, label='Training Accuracy')
ax3.plot(range(1, num_epochs+1), val_accuracies, label='Validation Accuracy')
ax3.set_xlabel('Epoch')
ax3.set_ylabel('Accuracy')
ax3.legend()
ax3.set_title('Training and Validation Accuracies')

plt.subplots_adjust(hspace=0.4)
plt.savefig('training_metrics_V2weights_pretrained.png')
plt.show()
```

5. <u>train_to_photos_custom_multy</u>

This .py script is used to train custom face recognition convolutional neural network (CNN) simple model with 3 convolutional layers, batch normalization, ReLU activations, and max pooling, followed by a fully connected layer, in which the final output logits serve the purpose of live face recognition (classification). The training procedure – steps are almost the same with the pretrained model described before. The differences lie on:

→ the number of epochs = 10 vs 5 epochs for the pretrained (as a simple model with 3 layers compared to the 48 layers of ResNet50 requires a higher number of epochs to achieve sufficient accuracy).

→ The definition of 'CustomModel' class: More specifically, the model's class is derived from the nn.Module class, which is the base class for all neural network modules in PyTorch.

→ We implemented new lines where the scripts search the file for previously saved model with predefined name. if it finds such model it loads its weight into the CNN and continues the train from there. If it doesn't find the file it starts the training with no weights.

→Also in this script we calculated f1 score and accuracy using the true positive, true negative ,false positive and false negative as a way to learn and experiment.

1. The model architecture consists of three convolutional layers (conv1, conv2, conv3) followed by batch normalization layers (bn1, bn2, bn3), and a fully connected layer (fc).

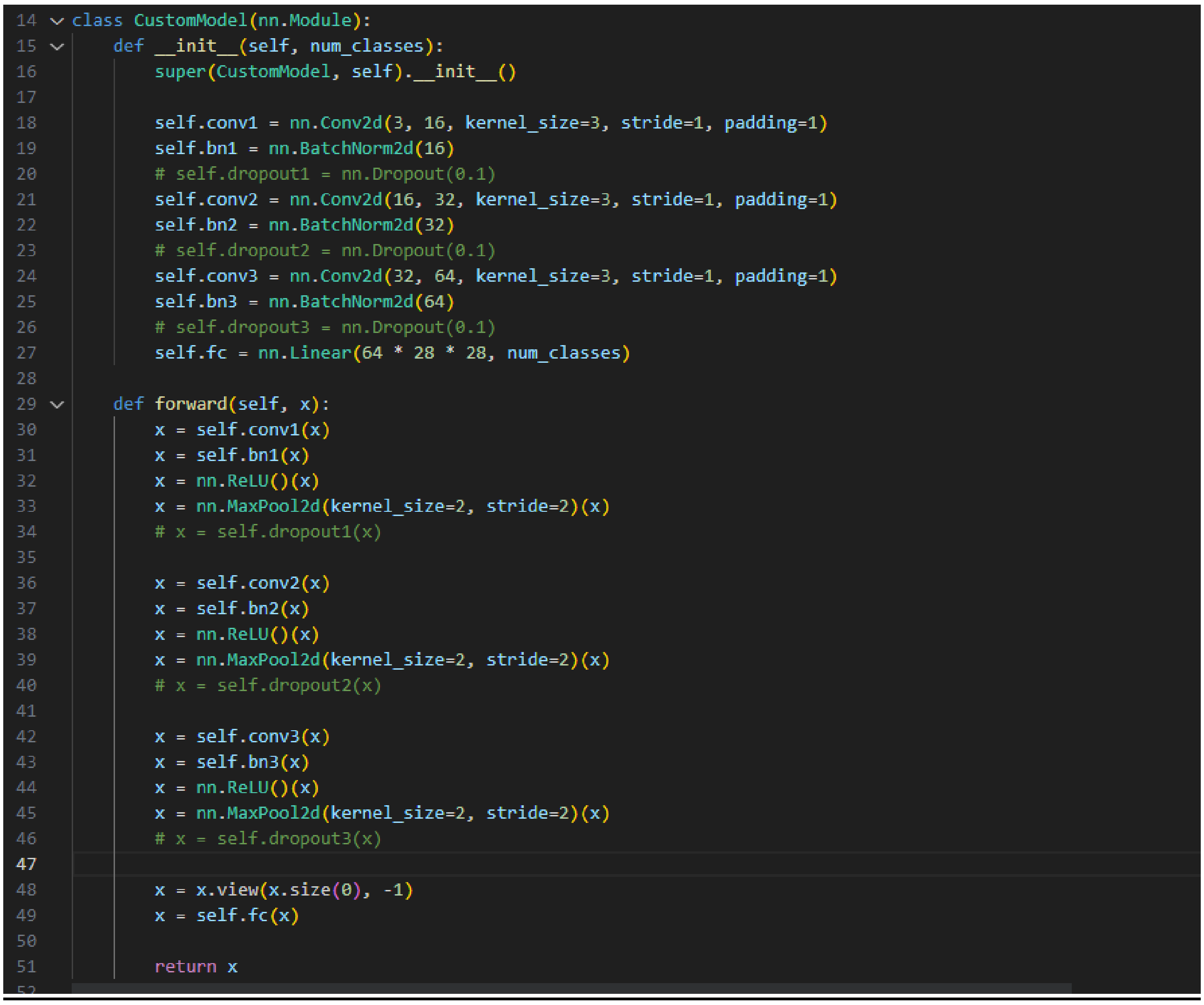
   * Batch normalization is a technique commonly used in deep neural networks, particularly in convolutional neural networks (CNNs), to improve the training and performance of the model. It aims to address the problem of internal covariate shift and stabilize the learning process.

   Internal covariate shift refers to the change in the distribution of intermediate layer activations during training as the parameters of the previous layers are updated. This can make it difficult for subsequent layers to learn effectively since they must constantly adapt to the changing input distributions.

   Batch normalization mitigates internal covariate shift by normalizing the inputs to each layer across a mini-batch of training examples. It operates by normalizing the activations of a layer's inputs to have zero mean and unit variance. The normalization is performed independently for each feature dimension, and additional learnable parameters, called scale and shift parameters, are introduced to allow the model to learn the optimal representation.

2. The conv1 layer takes input images with 3 channels (RGB) and applies 16 filters of size 3x3 with a stride of 1 and padding of 1. This results in 16 output channels.
3. The bn1 layer performs batch normalization on the output of conv1.
4. The ReLU activation function is applied after each convolutional layer using nn.ReLU().
5. Max pooling is applied after each ReLU activation using nn.MaxPool2d with a kernel size of 2x2 and stride of 2. This reduces the spatial dimensions of the feature maps by a factor of 2.
6. This downsampling process is repeated for the next two convolutional layers (conv2 and conv3) with increasing number of output channels (32 and 64, respectively).
7. After the third convolutional layer, the feature maps are flattened using x.view(x.size(0), -1) to transform them into a 1-dimensional tensor.
8. Finally, the flattened tensor is passed through the fully connected layer (fc) to produce the output logits, which correspond to the predicted class scores.

9. The number of output classes is provided as an argument to the CustomModel constructor (num_classes). The size of the fully connected layer (fc) is determined by 64 * 28 * 28, assuming the input images have a spatial size of 28x28 after three rounds of max pooling (64 channels * [224 (image size) / 2^3 due to 3 times max pooling = 28] *28).

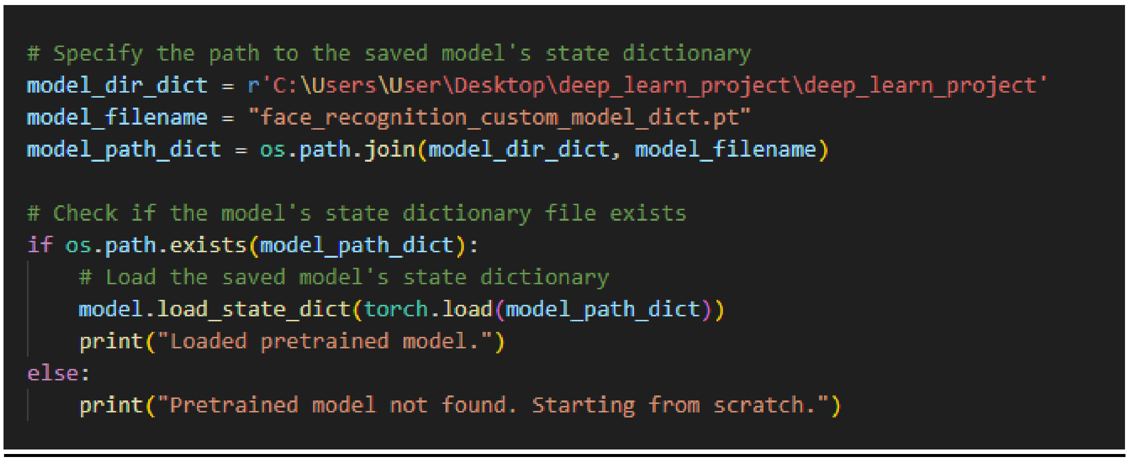


```python
class CustomModel(nn.Module):
    def __init__(self, num_classes):
        super(CustomModel, self).__init__()

        self.conv1 = nn.Conv2d(3, 16, kernel_size=3, stride=1, padding=1)
        self.bn1 = nn.BatchNorm2d(16)
        # self.dropout1 = nn.Dropout(0.1)
        self.conv2 = nn.Conv2d(16, 32, kernel_size=3, stride=1, padding=1)
        self.bn2 = nn.BatchNorm2d(32)
        # self.dropout2 = nn.Dropout(0.1)
        self.conv3 = nn.Conv2d(32, 64, kernel_size=3, stride=1, padding=1)
        self.bn3 = nn.BatchNorm2d(64)
        # self.dropout3 = nn.Dropout(0.1)
        self.fc = nn.Linear(64 * 28 * 28, num_classes)

    def forward(self, x):
        x = self.conv1(x)
        x = self.bn1(x)
        x = nn.ReLU()(x)
        x = nn.MaxPool2d(kernel_size=2, stride=2)(x)
        # x = self.dropout1(x)

        x = self.conv2(x)
        x = self.bn2(x)
        x = nn.ReLU()(x)
        x = nn.MaxPool2d(kernel_size=2, stride=2)(x)
        # x = self.dropout2(x)

        x = self.conv3(x)
        x = self.bn3(x)
        x = nn.ReLU()(x)
        x = nn.MaxPool2d(kernel_size=2, stride=2)(x)
        # x = self.dropout3(x)

        x = x.view(x.size(0), -1)
        x = self.fc(x)

        return x
```

Due to overfitting of our CNN we also tried to use dropout. Dropout is a regularization technique commonly used in neural networks to prevent overfitting. Overfitting occurs when a model becomes too specialized in training data and performs poorly on unseen or test data. Dropout helps address this issue by reducing the interdependencies between units (neurons) in a neural network. During training, dropout randomly sets a fraction of the input units or activations to zero at each update step. This means that the affected units are "dropped out" or temporarily removed from the network for that particular iteration. The fraction of units to be dropped out is determined by a hyper parameter called the dropout rate, typically set between 0.1 and 0.5.

By dropping out units, dropout prevents the network from relying too much on specific units and encourages the network to learn more robust and generalizable representations. It acts as a form of ensemble learning, as the network effectively trains multiple subnetworks with shared weights but different random dropouts.

In our case dropout was causing our CCN to over generalize and dropped the confidence of the prediction very much. That's why we abandoned it.



```
# Specify the path to the saved model's state dictionary
model_dir_dict = r'C:\Users\User\Desktop\deep_learn_project\deep_learn_project'
model_filename = "face_recognition_custom_model_dict.pt"
model_path_dict = os.path.join(model_dir_dict, model_filename)

# Check if the model's state dictionary file exists
if os.path.exists(model_path_dict):
    # Load the saved model's state dictionary
    model.load_state_dict(torch.load(model_path_dict))
    print("Loaded pretrained model.")
else:
    print("Pretrained model not found. Starting from scratch.")
```

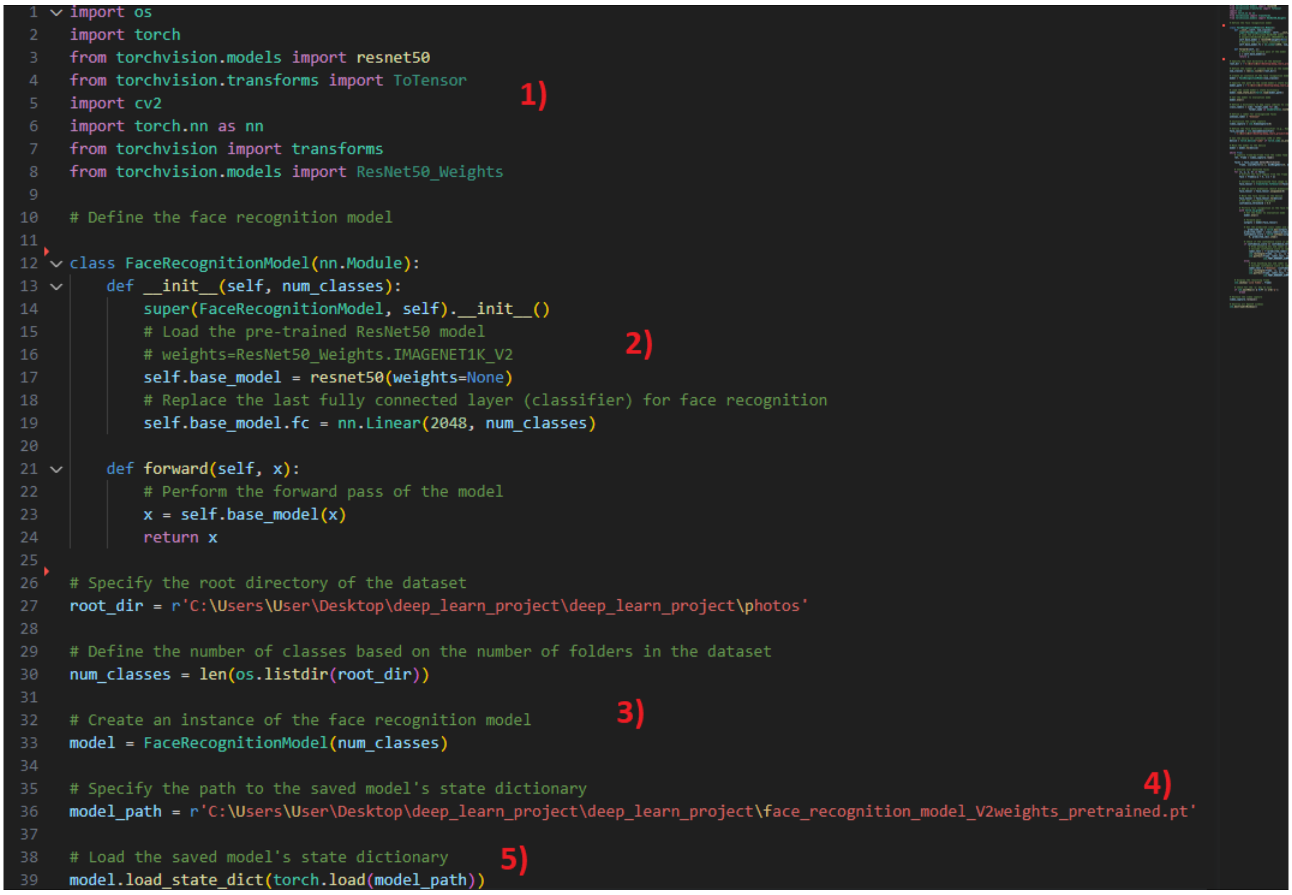
In the above figure the script search the file for previously saved model with predefined name. if it finds such model it loads its weight into the CNN and continues the train from there. If it doesn't find the file it starts the training with no weights.

6. <u>live_recognition_ver_pretrained</u>

Overall, this .py script allows real-time face recognition using a webcam and the pre-trained ResNet-50 model. It detects faces in the video feed, recognizes known faces, and labels them with their corresponding class names. Unknown faces are labeled as "Unknown." In our case we have a multi-classification problem with 3 classes (JohnT, HarisA and Unknown). Apparently, the target can be expanded to more classes. The implementation procedure and the main steps performed by the program are presented below:

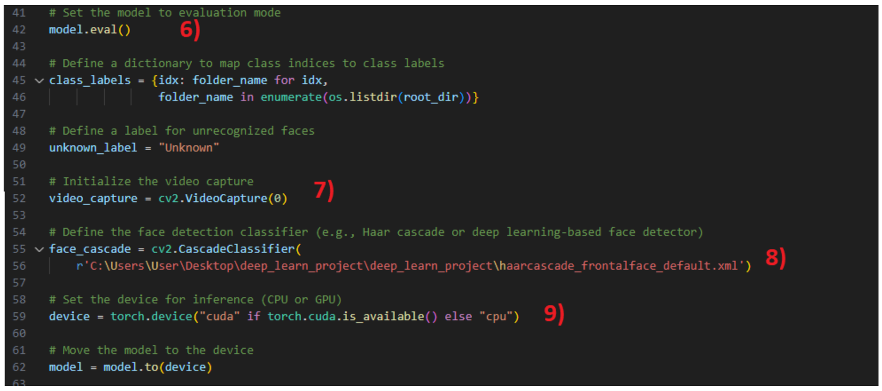
1. It imports the necessary libraries, including PyTorch, OpenCV (cv2), and torchvision.

2. It defines the face recognition model used (pre-trained ResNet-50 model)
3. It counts the folders of users and gives the model the number of classes
4. loads the saved state dictionary of the trained model, from a specified path.
5. Loads the weight into to CNN



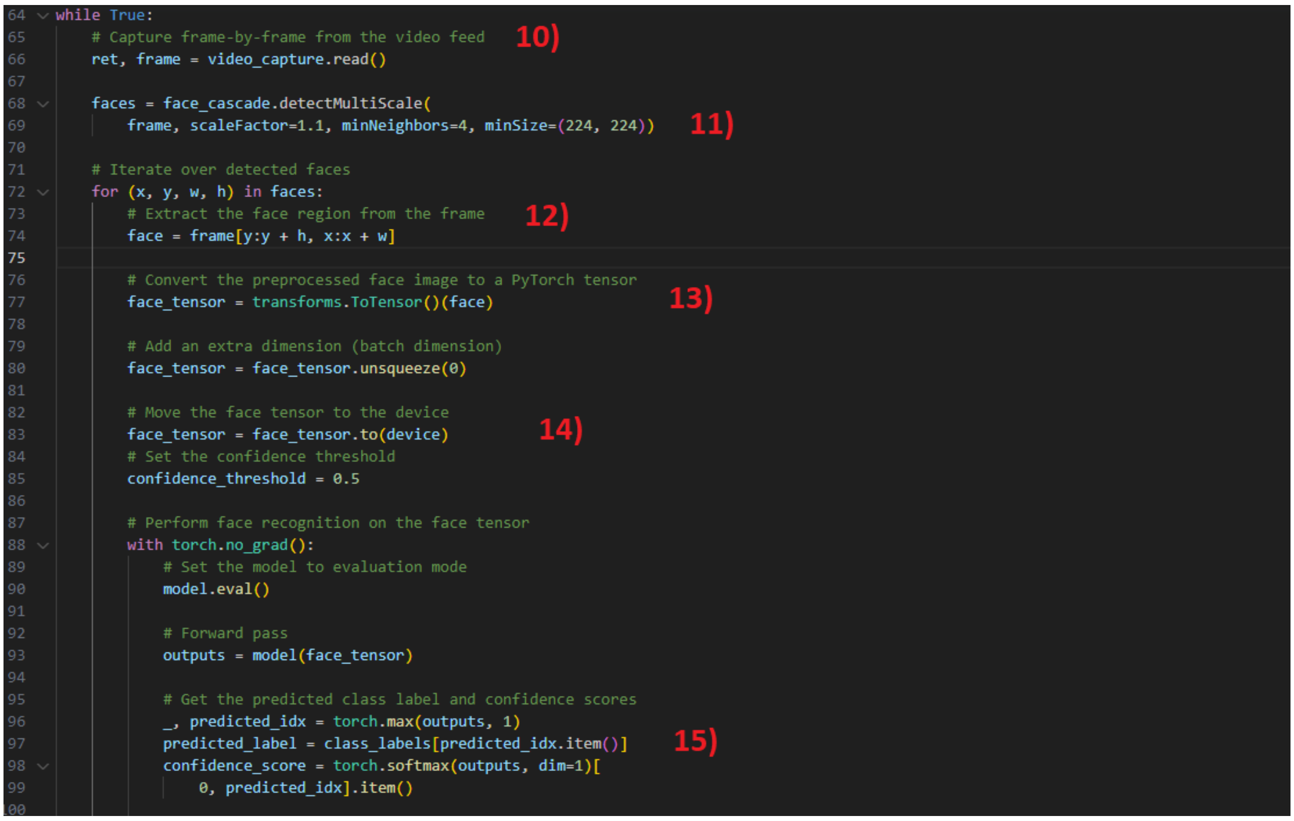
```python
import os
import torch
from torchvision.models import resnet50
from torchvision.transforms import ToTensor
import cv2
import torch.nn as nn
from torchvision import transforms
from torchvision.models import ResNet50_Weights

# Define the face recognition model

class FaceRecognitionModel(nn.Module):
    def __init__(self, num_classes):
        super(FaceRecognitionModel, self).__init__()
        # Load the pre-trained ResNet50 model
        # weights=ResNet50_Weights.IMAGENET1K_V2
        self.base_model = resnet50(weights=None)
        # Replace the last fully connected layer (classifier) for face recognition
        self.base_model.fc = nn.Linear(2048, num_classes)

    def forward(self, x):
        # Perform the forward pass of the model
        x = self.base_model(x)
        return x

# Specify the root directory of the dataset
root_dir = r'C:\Users\User\Desktop\deep_learn_project\deep_learn_project\photos'

# Define the number of classes based on the number of folders in the dataset
num_classes = len(os.listdir(root_dir))

# Create an instance of the face recognition model
model = FaceRecognitionModel(num_classes)

# Specify the path to the saved model's state dictionary
model_path = r'C:\Users\User\Desktop\deep_learn_project\deep_learn_project\face_recognition_model_V2weights_pretrained.pt'

# Load the saved model's state dictionary
model.load_state_dict(torch.load(model_path))
```

6. It sets the model to evaluation mode and defines class labels for the recognized faces.
7. It initializes the video capture from a webcam.
8. It defines the face detection classifier (haarcascade_frontalface).
9. It sets the device for inference (CPU or GPU) and moves the model to the specified device.



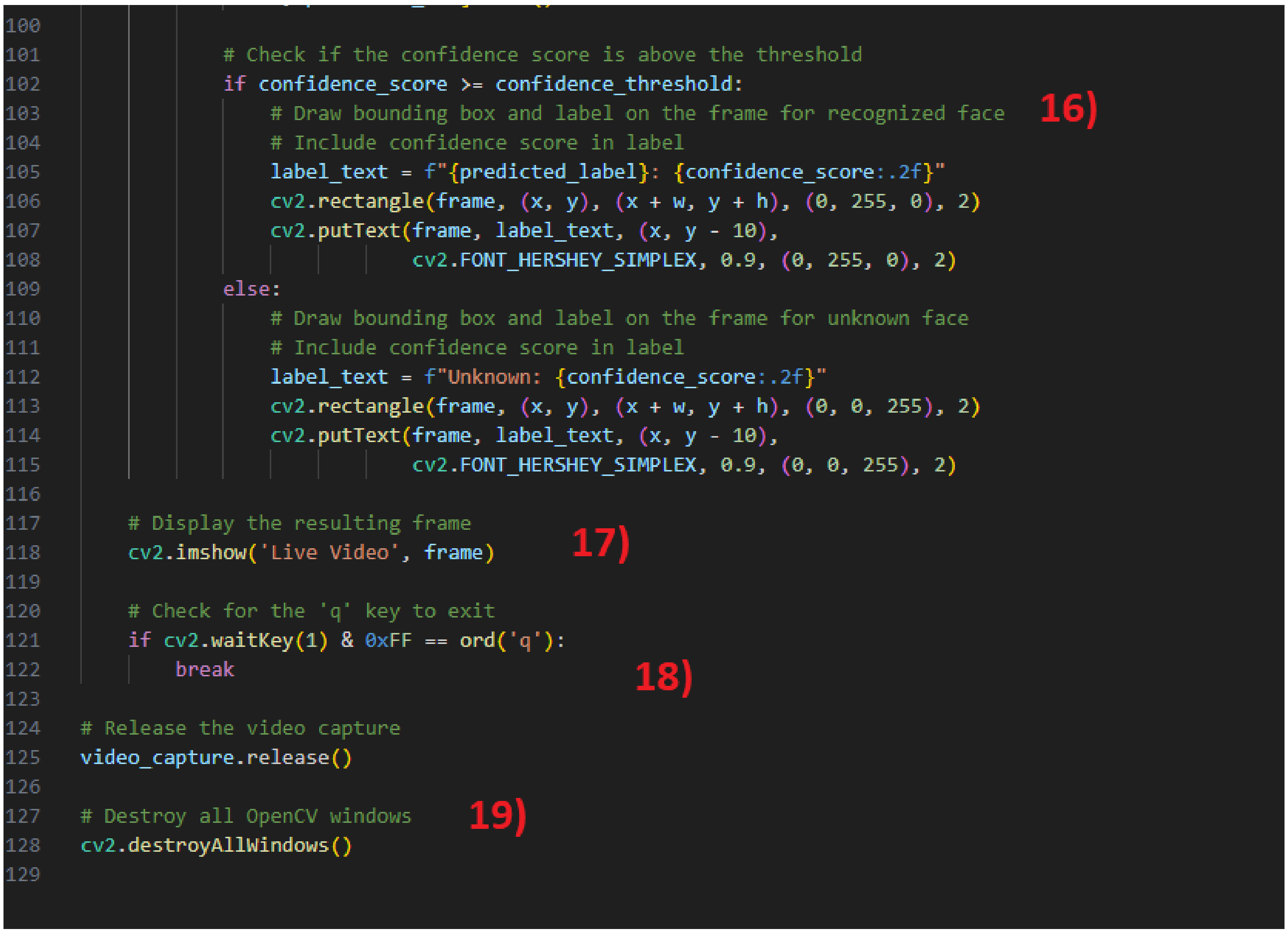
```python
# Set the model to evaluation mode
model.eval()

# Define a dictionary to map class indices to class labels
class_labels = {idx: folder_name for idx,
                folder_name in enumerate(os.listdir(root_dir))}

# Define a label for unrecognized faces
unknown_label = "Unknown"

# Initialize the video capture
video_capture = cv2.VideoCapture(0)

# Define the face detection classifier (e.g., Haar cascade or deep learning-based face detector)
face_cascade = cv2.CascadeClassifier(
    r'C:\Users\User\Desktop\deep_learn_project\deep_learn_project\haarcascade_frontalface_default.xml')

# Set the device for inference (CPU or GPU)
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

# Move the model to the device
model = model.to(device)
```

10. It enters a loop where it continuously captures frames from the video feed.
11. For each frame, it performs face detection using the face detection classifier.
12. For each detected face, it preprocesses the face image, converts it to a PyTorch tensor, and adds an extra dimension (batch dimension).
13. It moves the face tensor to the specified device.
14. It performs face recognition on the face tensor by passing it through the loaded model.
15. It retrieves the predicted class label and confidence score for the recognized face.



```python
while True:
    # Capture frame-by-frame from the video feed
    ret, frame = video_capture.read()

    faces = face_cascade.detectMultiScale(
        frame, scaleFactor=1.1, minNeighbors=4, minSize=(224, 224))

    # Iterate over detected faces
    for (x, y, w, h) in faces:
        # Extract the face region from the frame
        face = frame[y:y + h, x:x + w]

        # Convert the preprocessed face image to a PyTorch tensor
        face_tensor = transforms.ToTensor()(face)

        # Add an extra dimension (batch dimension)
        face_tensor = face_tensor.unsqueeze(0)

        # Move the face tensor to the device
        face_tensor = face_tensor.to(device)
        # Set the confidence threshold
        confidence_threshold = 0.5

        # Perform face recognition on the face tensor
        with torch.no_grad():
            # Set the model to evaluation mode
            model.eval()

            # Forward pass
            outputs = model(face_tensor)

            # Get the predicted class label and confidence scores
            _, predicted_idx = torch.max(outputs, 1)
            predicted_label = class_labels[predicted_idx.item()]
            confidence_score = torch.softmax(outputs, dim=1)[
                0, predicted_idx].item()
```

16. It draws a green bounding box (with specified design) and label on the frame for the recognized face, or for unknown faces if the confidence score is below a specified threshold.
17. It displays the resulting frame with the bounding boxes and labels.
18. It checks for the 'q' key press to exit the program.
19. Once the loop is exited, it releases the video capture and closes all OpenCV windows.



```python

            # Check if the confidence score is above the threshold
            if confidence_score >= confidence_threshold:
                # Draw bounding box and label on the frame for recognized face
                # Include confidence score in label
                label_text = f"{predicted_label}: {confidence_score:.2f}"
                cv2.rectangle(frame, (x, y), (x + w, y + h), (0, 255, 0), 2)
                cv2.putText(frame, label_text, (x, y - 10),
                            cv2.FONT_HERSHEY_SIMPLEX, 0.9, (0, 255, 0), 2)
            else:
                # Draw bounding box and label on the frame for unknown face
                # Include confidence score in label
                label_text = f"Unknown: {confidence_score:.2f}"
                cv2.rectangle(frame, (x, y), (x + w, y + h), (0, 0, 255), 2)
                cv2.putText(frame, label_text, (x, y - 10),
                            cv2.FONT_HERSHEY_SIMPLEX, 0.9, (0, 0, 255), 2)

        # Display the resulting frame
        cv2.imshow('Live Video', frame)

        # Check for the 'q' key to exit
        if cv2.waitKey(1) & 0xFF == ord('q'):
            break

    # Release the video capture
    video_capture.release()

    # Destroy all OpenCV windows
    cv2.destroyAllWindows()

```

7. <u>live_recognition_custom_cnn</u>

This .py script allows real-time face recognition using a webcam exactly the same as the previously described one It detects faces in the video feed, recognizes known faces, and labels them with their corresponding class names. Unknown faces are labeled as "Unknown." The only difference with the previous one, is that this live_recognition script is used to implement our own custom CNN model.

➢ Results

All models were tuned independently from the main scripts using smaller dataset due to computational limitations. The final parameters deemed appropriate to apply were:

- Optimizer: Adam ("Adaptive Moment Estimation") chosen between stochastic gradient descent (SGD), AdaGrad, or RMSprop
- Learning rate: 0.00001
- Loss function: CrossEntropyLoss
- Batch size: 128
- No of Epochs: 10 for our custom CNN (due to lighter architecture and 5 for ResNet50
- Activation function: ReLU (Rectified Linear Unit), in all models trained
- Batch Normalization applied

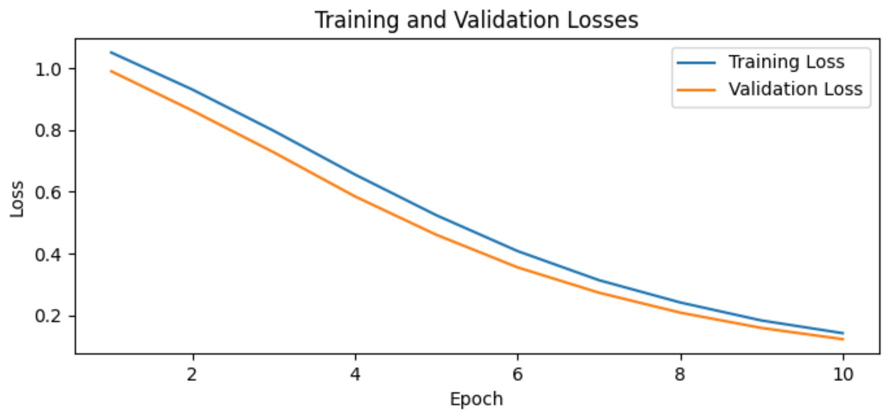
Regarding the time to train the 3 models, our records demonstrate that the 2 ResNet50 models took approximately 24 hours with 5 epochs, while for our custom CNN model the time elapsed to train was 10 hours for 10 epochs, that is half the time needed compared to ResNet50. It is worth mentioning, that as we increase the dataset the time needed to train ResNet50 increases greatly, compared to our own CNN model, mainly due to ResNet50 higher complexity. As mentioned above although we freeze the first 3 blocks the remaining 2 blocks still have 27 layers to pass from during training.

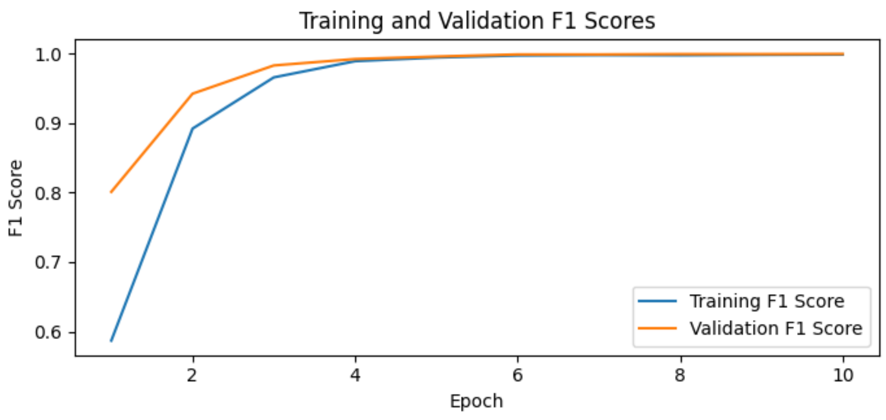
We tried to improve the parameters combination till ending up with a highly acceptable level of accuracy. All 3 models finally achieved high accuracy. As regards the pretrained model with weights derived from IMAGENET1K_V2, even if our dataset was very much smaller compared to IMAGENET, maybe the 27 layers of architecture left to be retrained and adjust, contributed to the final high accuracy. Their training and validation f1, loss and accuracy scores are presented per epoch in the following graphs. Nevertheless, it is worth mentioning that in the live face recognition assessment stage we noticed significantly lower accuracy, which might be due to:

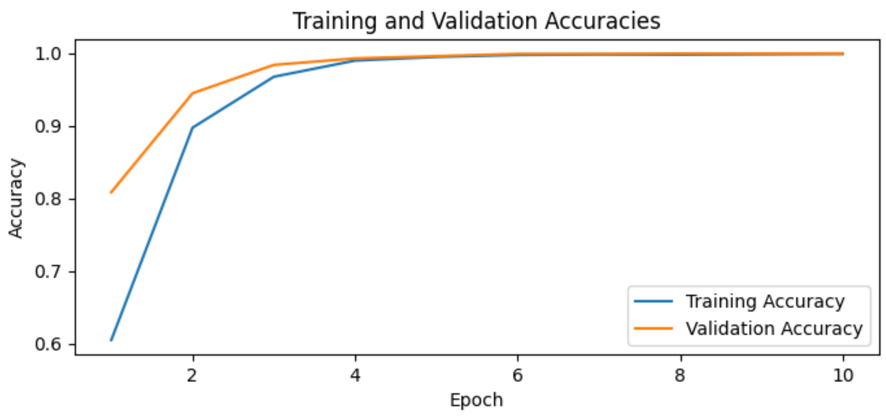
a) the small dataset used for training combined with
b) the highly effective (in models' performance) variations in our faces' characteristics over time (beard, hair etc) or the background colors, shadows, contrast, web camera specs etc.
c) a high minimum confidence threshold

- <u>**ResNet-50 architecture and weights derived from IMAGENET1K_V2**</u>

Already since the 3rd epoch the model reaches 100% accuracy, however due to the small training dataset we chose to continue with 5 epochs to avoid misclassified frames in live recognition phase.

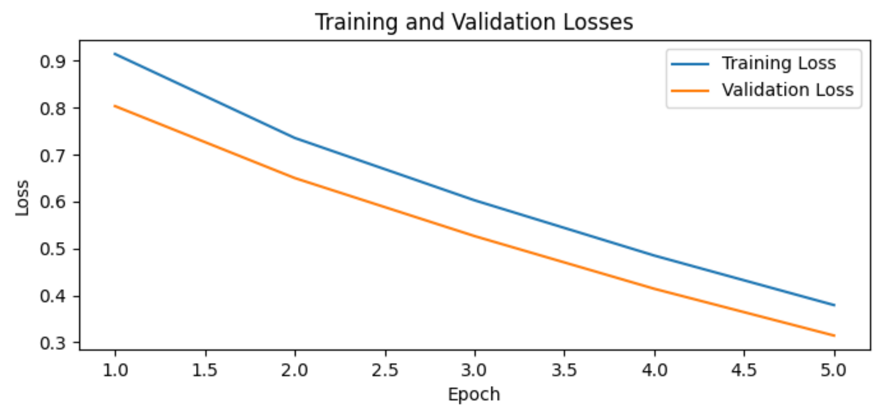


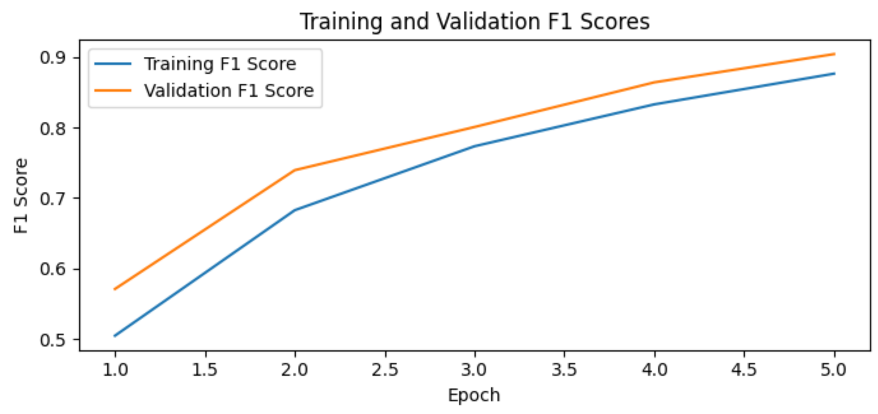


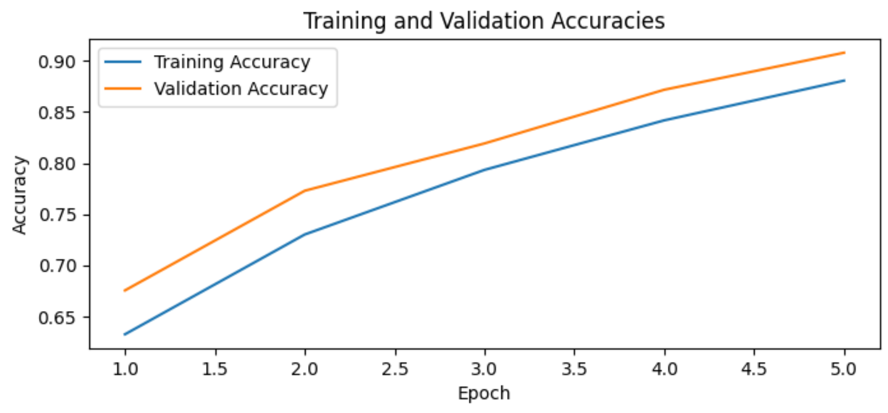


- **<u>ResNet-50 architecture and weights = None</u>**

Although in the training set the accuracy didn’t reach maximum level, in the 5th epoch we observe a maximum accuracy and minimum loss on the validation set.

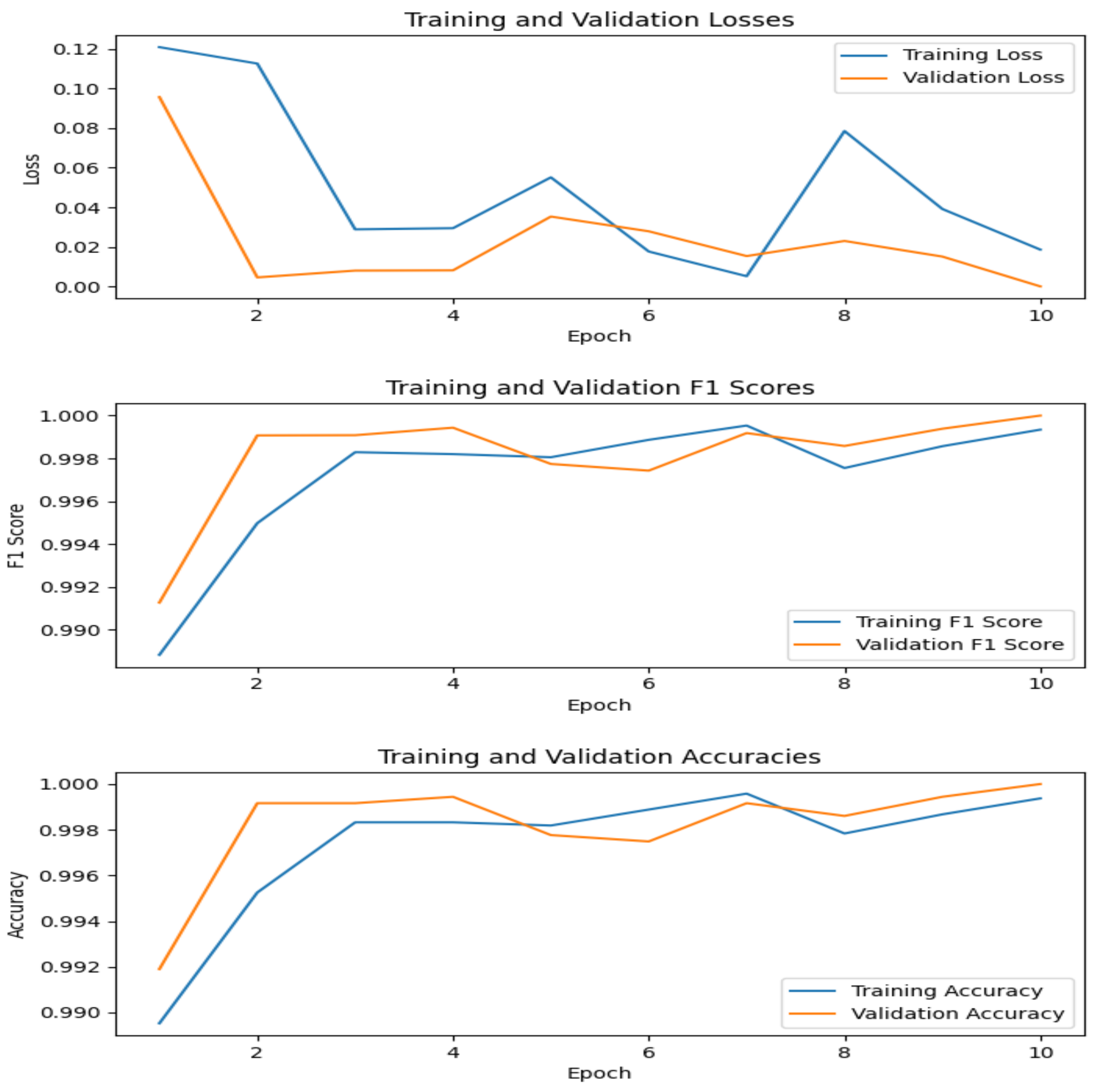




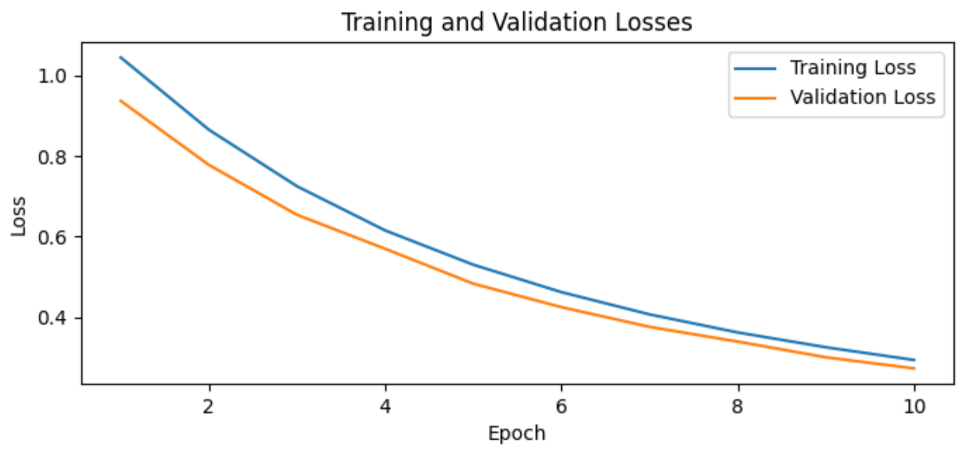


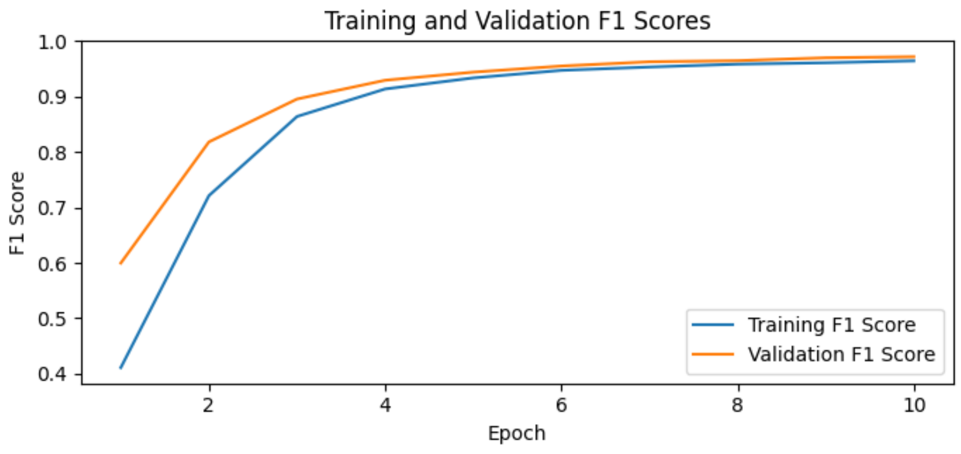
- <u>**Custom CNN model architecture**</u>

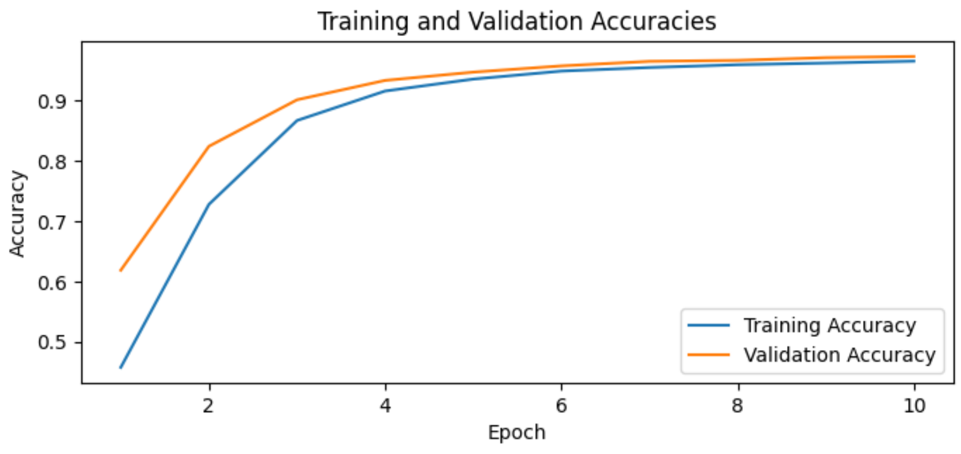
It is worth noting that while fluctuations in the training loss are observed, we see a decreasing trend in the loss over time, indicating that the model is learning and converging towards an optimal solution. We could adjust the learning rate, use different optimization techniques, or revisit the model architecture. Maybe the learning rate is high and the optimization process overshoots the minimum, causing the loss to increase temporarily. However, we achieve performance maximisation in 10 epochs.



The results after adjusting the learning rate to 0.000001 from 0.001 confirm our previous hypothesis for the learning rate. That is, now we observe a much more smooth line from epoch to epoch without fluctuations.







➢ Conclusions

After carefully reviewing the development procedure, the problem’s specific requirement and targets, and experimenting with hyperparameters tuning and live assessment of the 3 trained face recognition models we come to the following conclusions:

The final performance of the model depends on a combination set of many factors. The compound effect of many parameters as well as the difficulty and nature of the problem itself can significantly affect the performance of the model in real time. Also, it is remarkable that even if in static face classification the algorithm shows high accuracy, in live face recognition through camera, it can typically fail. As mentioned in the results this could be due to the dynamical changes in many factors such as the faces variations over time compared to training, the frames background characteristics or even the camera specs.

The goal when building a model of this kind should not be to influence the result by causing overfitting but to identify the parameters that influence it and try to create a properly trained model that in a long term is easily scalable and reproducible, in order to serve more complicated goals. Therefore, its a matter of trial and error to strike a balance between the dataset quantity and quality and the model parameters to achieve the desired performance. The parameters that jointly affect the models and their performance could be the following:

1. the nature of the problem: The very nature of the problem affects the type, quality, characteristics and quantity of data needed, the cost of collecting them and developing the desired model, the accuracy that the algorithm should have (minimum confidence threshold), provides an intuition for the initialization of its parameters.

2. the dataset size and quality. It should neither be too big and extensive nor too small. Also, all classes should have a balanced representation in the dataset. In our case we used a fairly small dataset due to computational difficulties and limitations, as ideally, we would like it to be at least 3-4 times larger. Nevertheless, due to the level of difficulty of the problem and other parameters that we adjusted in combination, the effectiveness of the models was preserved at acceptable levels. Furthermore the quality of the dataset is of high importance, that is in our case, the different angles, face characteristics, colors, contrasts, rotations and other variations and transformations. Data augmentation techniques, such as random cropping, rotation, flipping, or adding noise to the training data, can increase the diversity of the samples and improve the model's ability to generalize.

The parameters of the data augmentation process, such as the magnitude of rotation or the probability of flipping, can be tuned.

3. The number of classes in classification problems such as in our case (face recognition). As the number of classes increases, so does the need for more data to facilitate training of the algorithm. It is noticeable that for the “Unknown” class, the algorithm needs less training because in case it does not recognize the face as one of the two personalised classes (HarisA, JohnT) it has been coded to return the label “Unknown”. However, we clearly chose to train the models with balanced sample of “unknown” face images to facilitate generalization of the models and reduce overfitting.

4. Batch Size: The batch size determines the number of samples propagated through the network before the optimizer performs a parameter update. A larger batch size may lead to more stable updates but requires more memory, while a smaller batch size introduces more noise in the gradient estimates. It is typically chosen based on the available memory and computational constraints.

5. Network Architecture: The architecture of the CNN itself can be tuned, including the number and size of convolutional layers, pooling layers, fully connected layers, and their respective hyperparameters. The depth and width of the network, the size of the filters, and the number of channels can all have an impact on the model's capacity to learn and generalize. A more complex architecture with a large number of parameters can potentially capture intricate patterns and achieve higher accuracy but might be prone to overfitting, especially with limited training data.

6. Activation Functions: The choice of activation function for each layer can influence the network's ability to model complex non-linear relationships. Common choices include ReLU (Rectified Linear Unit), Leaky ReLU, sigmoid, and tanh. Experimenting with different activation functions may improve the network's performance.

7. Regularization Techniques: Regularization methods are used to prevent overfitting and improve the model's generalization. Techniques such as L1 or L2 regularization, dropout, or batch normalization can be applied to the layers of the CNN. Tuning the regularization strength or dropout rate can have an impact on the model's performance.

8. Pooling Strategies: Pooling layers downsample the spatial dimensions of the feature maps. Parameters such as the pooling size, stride, and type (e.g., max pooling, average pooling) can be adjusted to control the level of spatial information preservation and generalization capacity of the network.

9. Initialization Methods: The initial values of the network's weights can affect how quickly the model converges and the quality of the solutions found. Popular weight initialization techniques include random initialization, Xavier initialization, and He initialization. Choosing the appropriate initialization method can facilitate training.

10. Optimizer: The choice of optimizer affects how the model's parameters are updated during training. Different optimizers have different characteristics and adaptability to different problem domains. For instance, optimizers like Adam, RMSprop, and AdaGrad have adaptive learning rate mechanisms, while plain SGD has a fixed learning rate. The optimizer influences the speed and stability of convergence, as well as the model's ability to generalize. Choosing the right optimizer depends on the problem, dataset, and often requires experimentation to find the best fit.

11. Learning Rate: The learning rate determines the step size taken during parameter updates. A high learning rate can cause the model to converge quickly but risk overshooting the optimal solution or even diverging. Conversely, a low learning rate may lead to slow convergence or getting stuck in suboptimal solutions. The learning rate needs to be carefully chosen to achieve a balance between convergence speed and stability.

12. Learning Rate Schedule: Instead of using a fixed learning rate throughout training, a learning rate schedule can be employed to adjust the learning rate over time. Techniques such as step decay, exponential decay, or cyclical learning rates can be explored to find an optimal learning rate strategy.

13. Number of Epochs: The number of epochs determines how many times the model iterates over the entire training dataset. Too few epochs may result in underfitting, where the model hasn't learned enough from the data, leading to poor performance. On the other hand, too many epochs can lead to overfitting, where the model becomes too specialized to the training data and performs poorly on

unseen data. The optimal number of epochs varies depending on the complexity of the task and the size of the dataset.

14. Early Stopping: Early stopping is a technique that monitors the validation loss during training and stops training when the validation loss starts to increase, thus preventing overfitting. The patience parameter, which determines how many epochs to wait before stopping, can be tuned.

The performance of the model is influenced by the interplay of these factors. Optimal hyperparameter values and architecture depend on the specific problem and dataset. It often requires a combination of experience, domain knowledge, and experimentation to find the best configuration. Last but not least, it is important to mention that as the complexity and problem requirements increase, so will the corresponding required resources (financing for computing power, suitable dataset etc.) to cope with them.